

CONFERENCE PROCEDINGS

USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems

New Orleans, Louisiana January 16–20, 1995



The UNIX® and Advanced Computing Systems Professional and Technical Association

WINTER

The USENIX Association

Proceedings of the 1995 USENIX Technical Conference

January 16 - 20, 1995 New Orleans, Louisiana, USA For additional copies of these proceedings, write:

USENIX Association 2560 Ninth Street, Suite 215 Berkeley, CA 94710 USA

The price is \$30 for members and \$39 for nonmembers.

Outside the USA and Canada, please add
\$20 per copy for postage (via air printed matter).

Past USENIX Technical Conferences

1994 Summer Boston	1988 Summer San Francisco
1994 Winter San Diego	1988 Winter Dallas
1993 Summer Cincinnati	1987 Summer Phoenix
1993 Winter San Diego	1987 Winter Washington, DC
1992 Summer San Antonio	1986 Summer Atlanta
1992 Winter San Francisco	1986 Winter Denver
1991 Summer Nashville	1985 Summer Portland
1991 Winter Dallas	1985 Winter Dallas
1990 Summer Anaheim	1984 Summer Salt Lake City
1990 Winter Washington, DC	1984 Winter Washington, DC
1989 Summer Baltimore	1983 Summer Toronto
1989 Winter San Diego	1983 Winter San Diego

1995 © Copyright by The USENIX Association All Rights Reserved.

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 1-880446-67-7



TABLE OF CONTENTS

USENIX 1995 Technical Conference January 16-20, 1995 New Orleans, Louisiana

Opening Remarks	
Peter Honeyman, CITI, University of Michigan	
Keynote Address: Ubiquitous Computing	
Mark Weiser, Xerox Palo Alto Research Center	
De	CD
B	SD
Wednesday, 11:00 - 12:30	Session Chair: John T. Kohl, Atria Software
	Session Chair Goin 1. Ixom, Ixtra Software
Portals in 4.4BSD	
W. Richard Stevens, Consultant; Jan-Simon Pendry, Sequ	ent Corporation
Aju John, Digital Equipment Corporation	
Union Mounts in 4.4BSD-Lite	
Jan-Simon Pendry, Sequent Corporation; Marshall Kirk	
BMA CC	CTOR
MASS	STORE
Wednesday, 2:00 - 3:30	Session Chair: Charles J. Antonelli, CITI,
	University of Michigan
	o management
Evaluation of Design Alternatives for a Cluster File Syste	em
Murthy Devarakonda, Ajay Mohindra, Jill Simoneaux, W	
Jonathan S. Goldick, Kathy Benninger, Christopher Kirby Pittsburgh Supercomputing Center	y, Christopher Maher, and Bill Zumach,
1 moonigh supercompaning center	
RAMA: Easy Access to a High-Bandwidth Massively Pa	rallel File System 59
Ethan L. Miller, University of Maryland, Baltimore Coun	ty; Randy H. Katz, University of California, Berkeley
POTPO	URRI I
rorro	UKKII
W. J J 4.00 5.20	
Wednesday, 4:00 - 5:30	Session Chair: Greg Minshall, Novell, Inc.
Implementing Real Time Packet Forwarding Policies Usi	ng Streams
Ian Wakeman, Atanu Ghosh, Jon Crowcroft, University C Lawrence Berkeley Laboratory	ouege London; van Jacobson, Sally Floyd,
Zamence Derketey Euboratory	
Scaling the Web of Trust: Combining Kerberos and PGP	to Provide Large Scale Authentication
Jeffrey I. Schiller, Derek Atkins, MIT	

Puneet Kumar, M. Satyanarayanan, Carnegie Mellon University

OBJECTS

Thursday, 9:00 - 10:30	Session Chair: Richard Draves, Microsoft Research
OODCE: A C++ Framework for the OSI John Dilley, Hewlett-Packard	F Distributed Computing Environment
Mach-US: UNIX On Generic OS Object S J. Mark Stevenson, Carnegie Mellon Uni	Servers
Events in an RPC Based Distributed Sys Jim Waldo, Ann Wollrath, Geoff Wyant, S	tem
	POTPOURRI II
Thursday, 11:00 - 12:30	Session Chair: Lori Grob, Chorus Systèmes
Turning the AIX Operating System into Jacques Talbot, BULL	an MP-capable OS143
A Flash-Memory Based File System Atsuo Kawaguchi, Shingo Nishioka, Hir	oshi Motoda, Hitachi, Ltd.
TRON: Process-Specific File Protection Andrew Berman, Virgil Bourassa, Erik S	for the UNIX Operating System
THE	CY COME FROM PALO ALTO
Thursday, 2:00 - 3:30	Session Chair: Phil Winterbottom, AT&T Bell Laboratories
SIFT – a Tool for Wide-Area Informatio Tak W. Yan, Hector Garcia-Molina, Stan	on Dissemination
Performance Implications of Multiple P Jeffrey C. Mogul, Joel F. Bartlett, Rober Western Research Laboratory	ointer Sizes
Idleness is Not Sloth	elin, Tim Sullivan, and John Wilkes, Hewlett-Packard Laboratories
	LIBRARIES
Friday, 9:00 - 10:30	Session Chair: Douglas Orr, University of Utah
Libckpt: Transparent Checkpointing und James S. Plank, Micah Beck, Gerry Kin	der UNIX213 gsley, University of Tennessee; Kai Li, Princeton University
	cally-Linked Programs
DP: A Library for Building Portable, Ro David M. Arnow, Brooklyn College, CU	eliable Distributed Applications

FILE SYSTEMS

Friday, 11:00 - 12:30	Session Chair: Noemi Paciorek, Horizon Resea	rch
File System Logging versus Clusteri	ng: A Performance Comparison	249
Margo Seltzer, Keith A. Smith, Harvo Venkata Padmanabhan, University o	ard University; Hari Balakrishnan, Jacqueline Chang, Sara McMains,	
Metadata Logging in an NFS Server.		265
Uresh Vahalia, EMC Corporation; C EMC Corporation	Cary G. Gray, Abilene Christian University; Dennis Ting,	203
Heuristic Cleaning Algorithms in Lo	g-Structured File Systems	277
Trevor Blackwell, Jeffrey Harris, Ma	rgo Seltzer, Harvard University	277
	ARCHITECTURE	
Friday, 2:00 - 3:30	Session Chair: Bob Gray, US WEST Techn	ologie
The New Jersey Machine-Code Tool Norman Ramsey, Bell Communicatio	kit ns Research; Mary F. Fernandez, Princeton University	289
ATOM: A Flexible Interface for Buil Alan Eustace, Amitabh Srivastava, D	ding High Performance Program Analysis Toolsigital Equipment Corporation, Western Research Laboratory	303
Adaptable Binary Programs		315
Susan L. Graham, Steven Lucco, Rob	ert Wahbe, University of California, Berkeley	

ACKNOWLEDGMENTS

Program Chair

Peter Honeyman

Center for Information Technology Integration (CITI) University of Michigan

Program Committee

Charles J. Antonelli, CITI, University of Michigan David Bachmann, IBM, Austin
Cecelia D'Oliviera, Information Systems, MIT
Richard Draves, Microsoft Research
Bob Gray, US WEST Technologies
Lori Grob, Chorus Systèmes
Peter Honeyman, CITI, University of Michigan
John T. Kohl, Atria Software
Greg Minshall, Novell, Inc.
Douglas Orr, Universities of Utah and Michigan
Noemi Paciorek, Horizon Research
Phil Winterbottom, AT&T Bell Laboratories

Readers

Sarr Blumson, CITI, University of Michigan Keith Bostic, CSRG, UCB Thomas A. Cargill, Consultant Po-Yung Chang, University of Michigan Geoffrey Clemm, Bellcore Fred Douglis, AT&T Bell Laboratories Mike Durian, Bellcore Darren Hardy, University of Colorado Larry Huston, CITI, University of Michigan Mike Jones, Microsoft Research Bill Joy, Sun Microsystems Chet Juszcak, Digital Equipment Corporation Berry Kercheval, Xerox PARC Jim Lipkis, Chorus Systèmes Darrell Long, UCSC Kirk McKusick, Consultant and Author Jeffrey Mogul, Digital Equipment Corporation, Western Research Laboratory Dan Muntz, CITI, University of Michigan Bob Oesterlin, IBM Brian Pawlowski, Network Appliance Corporation Tom Poindexter, Consultant Dave Presotto, AT&T Bell Laboratories Sushila R Subramanian, Naval Research Laboratory Jim Rees, CITI, University of Michigan Peter Reiher, UCLA Margo Seltzer, Harvard University Mark Smith, University of Michigan

Michael T. Stolarchuk, CITI, University of Michigan
Bennet Yee, Carnegie-Mellon University
Wesley Craig, University of Michigan
Atul Prakash, University of Michigan
Bill Doster, University of Michigan
Greg Rose, Sterling Software
Christopher Small, Harvard University
Carl Staelin, Hewlett Packard Laboratories
Brent Welch, Xerox PARC
Ben Zorn, University of Colorado

Proceedings Production

Carolyn S. Carr, USENIX Association Malloy Lithographing, Inc.

Invited Talks

Brent Welch, Xerox PARC
Ed Gould, Digital Equipment Corporation

Tutorial Program

Daniel V. Klein, USENIX Association

Works in Progress Coordinator

Peg Shafer, BBN

USENIX Board Liaison

Lori Grob, Chorus Systèmes

Administration

The USENIX Association Staff

Terminal Room

Gretchen Phillips, University of Buffalo

The GURU is In Coordinator

Ed Gould, Digital Equipment Corporation

Vendor Display

Peter Mui, USENIX Association

USENIX Marketing Director

Zanna Knight, USENIX Association

USENIX Executive Director

Ellie Young, USENIX Association

USENIX Meeting Planner

Judy DesHarnais, USENIX Association

Preface

Welcome to the 1995 USENIX Conference!

A USENIX conference is the result of many hands working together, and many thanks are due, but the first salute is to you: the attendees. Welcome to New Orleans! Thanks for coming!

Thanks as well to the 186 authors from 11 countries who submitted 80 papers for consideration. Three-quarters of the papers submitted were from the U.S. About 60% of the papers were from educational institutions, with the remainder coming from commerce and a handful from government labs. Each paper was read by an average of five reviewers; we selected 27 papers for presentation. The program committee is particularly pleased at the strength and diversity of the technical program.

Alongside the technical track, the 1995 USENIX conference offers a track of invited talks; thanks to Brent Welch and Ed Gould, the coordinators. This year's invited talks are especially timely and exciting. Together, the technical and invited talks offer an unparalleled opportunity to gain perspective on the state of the art and future directions in modern computing systems.

Thanks as well to Dan Klein and the tutorial selection committee for assembling an impressive collection of valuable and fascinating courses of instruction; to Peg Schafer for organizing the works-in-progress session; and to the USENIX staff who work so effectively to make life so easy for program chairs. I am especially grateful to Ellie Young and Judy DesHarnais, expert professionals at the top of their fields. I am deeply grateful to Terri Saarinen, Kati Bauer, and (especially!) Chris Tanis for making the program committee activities run smoothly at CITI.

Finally, I thank the members of the program committee and the external reviewers (especially Berry Kercheval and Chris Small) for volunteering their time and energy to help make the technical track of the 1995 USENIX conference a strong one. Please join me in showing our gratitude to all the folks who have contributed to the success of the New Orleans conference. And let the bon temps rouler!

Peter Honeyman, Program Chair

AUTHOR INDEX

Arnow, David M.	235	Lucco, Steven	315
Atkins, Derek	83	Maher, Christopher	47
Balakrishnan, Hari	249	Mayo, Robert N.	187
Bartlett, Joel F.	187	McKusick, Marshall Kirk	25
Beck, Micah	213	McMains, Sara	249
Benninger, Kathy	47	Miller, Ethan L.	59
Berman, Andrew	165	Mogul, Jeffrey C.	187
Blackwell, Trevor	277	Mohindra, Ajay	35
Bosch, Peter	201	Motoda, Hiroshi	155
Bourassa, Virgil	165	Nishioka, Shingo	155
Chang, Jacqueline	249	Padmanabhan, Venkata	249
Chang, Wei-Chau	225	Pendry, Jan-Simon	1, 25
Crowcroft, Jon	71	Plank, James S.	213
Devarakonda, Murthy	35	Ramsey, Norman	289
Dilley, John	107	Satyanarayanan, M.	95
Eustace, Alan	303	Schiller, Jeffrey I.	83
Fernandez, Mary F.	289	Selberg, Eric	165
Floyd, Sally	71	Seltzer, Margo	249, 277
Garcia-Molina, Hector	177	Simoneaux, Jill	35
Ghosh, Atanu	71	Smith, Keith A.	249
Goldick, Jonathan S.	47	Srivastava, Amitabh	187, 303
Golding, Richard	201	Staelin, Carl	201
Graham, Susan L.	315	Stevens, W. Richard	1
Gray, Cary G.	265	Stevenson, J. Mark	119
Harris, Jeffrey	277	Sullivan, Tim	201
Ho, W. Wilson	225	Talbot, Jacques	143
Jacobson, Van	71	Tetzlaff, William H.	35
John, Aju	11	Ting, Dennis	265
Julin, Daniel P.	119	Vahalia, Uresh	265
Katz, Randy H.	59	Wahbe, Robert	315
Kawaguchi, Atsuo	155	Wakeman, Ian	71
Kendall, Samuel C.	131	Waldo, Jim	131
Kingsley, Gerry	213	Wilkes, John	201
Kirby, Christopher	47	Wollrath, Ann	131
Kumar, Puneet	95	Wyant, Geoff	131
Leung, Lilian H.	225	Yan, Tak W.	177
Li, Kai	213	Zumach, Bill	47

BSD

Session Chair: John T. Kohl, Atria Software

NOTES

Portals in 4.4BSD

W. Richard Stevens Consultant

Jan-Simon Pendry Sequent UK

Abstract

Portals were added to 4.4BSD as an experimental feature and are in the publicly available 4.4BSD-Lite distribution. Portals provide access to alternate file types or devices using names in the normal filesystem that a process just opens. For example, an open of /p/tcp/foo.com/smtp returns a TCP socket descriptor to the calling process that is connected to the SMTP server on the specified host. By providing access through the normal filesystem, the calling process need not be aware of the special functions necessary to create a TCP socket and establish a TCP connection. This makes TCP connections, for example, available to programs such as Awk, Tcl, and shell scripts.

This paper describes the implementation of portals in 4.4BSD as another type of filesystem and provides some examples.

1. Introduction

The Unix paradigm of open, read, write, lseek, and close has worked well for many years across a wide range of devices. Having devices share the same namespace as regular files is one of the many things Unix did right. Terminals and networks, however, have always been unique and present an interesting set of problems.

One problem with terminal devices is the wide variety of modems that connect to them, and each modem's unique command string. This has led to the duplication of dialing code among various programs, such as tip, cu, and UUCP.

The problem with networks is that they have their own unique set of functions to create and manipulate network end points. Whether using the sockets programming interface or TLI, the code to establish a network end point differs greatly from the code for normal file I/O [Stevens 1990]. While the C programmer can easily access network connections

using either sockets or TLI, it has been a problem to access network connections using scripting languages such as a shell or Awk. One solution, chosen by Perl [Wall and Schwartz 1991] is to place all the socket calls into the language. While possible, this has the unfortunate side effect of increasing the size and complexity of the language.

The implementation of portals in 4.4BSD is an attempt to hide the complexity of things such as network connections within a system-wide portal daemon that is accessible to *any* process through the open function.

2. Descriptor Passing

Modern versions of Unix have supported *descriptor passing* between unrelated processes for many years. 4.2BSD was the first system to support this, circa 1983, although there were bugs in the initial implementation, which were fixed with 4.3BSD (1986). System V Release 3.2 (1988) also provided a similar capability, and a cleaner interface was provided with System V Release 4 (SVR4). Most commercial versions of Unix today, whether derived from BSD or System V, provide this capability. [Stevens 1990; 1992] provides examples of descriptor passing in both the BSD and System V worlds.

3. Connection Server

With the ability to pass an arbitrary descriptor between unrelated processes the idea of a *connection server* came about [Presotto and Ritchie 1985; 1990]. This server operates as follows:

- Any process creates a connection to the server, called a *stream pipe*. 4.3BSD implements stream pipes using Unix domain stream sockets, while the pipe system call is used under SVR4.
- The server receives notification that a new client has connected to it and a unique connection is created for each client. The BSD accept function

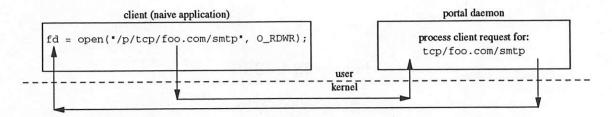


Figure 1: Overall design of portals.

creates the unique connections, as does the SVR4 connld streams module.

- The client sends a request to the server, often by writing some data to the stream pipe that the server reads.
- The server validates the request and does whatever is required. This might involve establishing a network connection to a remote system, or opening a modem device and dialing a remote system, for example. The purpose of the connection server is to place all these communications-dependent operations into a central user-mode server instead of scattering the same operations through numerous applications, or instead of trying to place them into the kernel.
- The server uses descriptor passing to return a descriptor to the client. The client can read and write this descriptor, without involving itself in all the steps required to open the descriptor.

[Stevens 1992] provides a complete example of a connection server that dials a modem, along with some general purpose client-server connection functions that hide the differences between the BSD and SVR4 implementations. The <code>ipc(3X)</code> man page [AT&T 1990] documents another set of client-server connection functions, the ones described in [Presotto and Ritchie 1990]. As powerful as the concept of a connection server is, it has been used sparingly in the widely used releases of Unix.

One problem with the connection server model is that it cannot be used by *naive* applications. That is, applications that only call open, read, and write. To take advantage of a connection server the application must be coded to connect to the server, send a request, and receive a descriptor in reply. Connection servers cannot be used by scripting languages such as Awk, Tcl, and shell scripts.

4. Portals

Portals provide a generalized "open" capability, similar to a connection server, but they solve the problem of naive applications. Portals are accessed by the

normal open function. Since most applications (be it a scripting language or a binary application that we want to run but do not have the source code for) allow us to enter a pathname of our choice, and since portals are accessed by pathnames, this makes portals available to virtually *any* application.

For example a naive application calls open with /p/tcp/foo.com/smtp as the pathname argument and the kernel returns either a nonnegative descriptor or -1 if an error occurs. The application then reads and writes the descriptor. Figure 1 shows the basic design.

The pathname argument to open begins with the mount point of a portal filesystem (/p in all the examples in this paper). The kernel passes this to the portal daemon, as we describe in detail in the next section. The daemon does whatever it needs to for this client, which in this case involves calling gethostbyname to convert the name foo.com into one or more IP addresses, calling getservbyname to obtain the port number for the smtp service, calling socket, and then calling connect. The daemon either succeeds and passes a descriptor back to the client, or the client's call to open returns -1 with errno set to the appropriate error code.

There are a few key points in the implementation of portals in 4.4BSD.

- The only function of the portal code within the kernel is to pass the pathname argument from open to the portal daemon, along with the client's credentials, and to take the daemon's return value, either a descriptor or an error code, and pass it back as the return value from open. The kernel does not interpret the pathname argument at all, other than detecting the portal filesystem mount point. The interpretation of the string tcp, followed by a host name, followed by a service name, is handled by the daemon.
- The portal daemon is a user process that can easily be enhanced or replaced. As experience is obtained with portals, additional objects besides TCP connections can be added without changing the kernel.

The use of portals and the portal daemon is completely transparent to the application. As a trivial example,

```
$ cat < /p/tcp/noao.edu/daytime
Wed Jul 6 11:26:07 1994
```

opens a TCP connection to the standard daytime server on the host noao.edu, which writes back the current time and date and closes the connection.

4. Since the client's open goes through the kernel before the request is passed to the portal daemon, the kernel provides the client's credentials to the server: the client's effective user ID, effective group ID, and supplementary group IDs. This information is passed to the daemon, in addition to the pathname from open. This lets the server implement any type of access restrictions that it desires, based on the client's identity.

5. Related Work

Ideas similar to portals have appeared in numerous operating systems over the past decade. We mention these briefly here since space prevents a detailed comparison of the various implementations.

The 4.2BSD manual [Joy et al. 1983] defined the portal system call, with seven arguments, and a footnote that it was not implemented in 4.2BSD.

[Bershad and Pinkerton 1988] describe watchdogs: user-level processes that can be called whenever a specified file or directory is opened.

The Sprite operating system provides pseudo devices [Welch and Ousterhout 1988]. It allows a user-level process to emulate a file or device.

Plan 9 [Presotto and Winterbottom 1993] represents network connections using *protocol devices*. All protocol devices look identical so user programs contain no network-specific code.

6. Implementation of Portals

Portals are implemented as a new filesystem type within the 4.4BSD kernel. When the system is initialized a portal daemon is normally started. The daemon performs the following steps:

- Creates a Unix domain stream socket, which we call listenfd.
- 2. Calls listen to allow incoming connection requests to be accepted on the socket.
- 3. Calls the mount function to mount the portal filesystem at a specified location in the filesystem

- (the mount point), typically /p. Additional portal-specific arguments to the mount function are the socket descriptor of the listening socket created by the server, and the server's process ID.
- The portal daemon then goes to sleep in an accept function, waiting for some process to open a pathname that begins with the portal mount point.

These four steps are summarized in Figure 2.

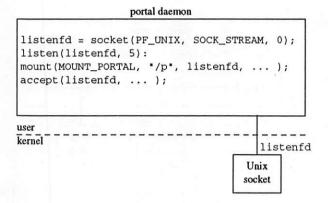


Figure 2: Initialization of portal daemon.

Within the vnode architecture of 4.4BSD (similar to the vnode architecture described in [Welch 1994]), the portal filesystem provides 11 filesystem (or *vfs*) operations: mount, unmount, statfs, and so on. About 40 file (or *vnode*) operations are also defined for each filesystem type: open, close, lookup, and so on.

Whenever a process (which we call the client) opens a pathname that begins with the portal mount point, the kernel passes control to the vnode lookup function defined for the portal filesystem: portal_lookup. The following steps take place.

- portal_lookup allocates a new vnode structure and the remainder of the pathname (whatever follows the mount point) is saved in the structure. For example, if a process opens the pathname /p/tcp/foo.com/smtp, the string tcp/foo.com/smtp is saved in the vnode structure.
- This new vnode is returned by portal_lookup and it becomes an argument to portal_open, which is the next kernel function called to process the open function.
- portal_open creates two new Unix domain stream sockets: the first we call clifd, and the second we call connfd. The second of these sockets is created and returned as the new

descriptor by the accept pending on the portal daemon's listening Unix socket. The first socket (clifd) is connected to connfd, as though both had been created by the socketpair function. This creates a full-duplex pipe between clifd and connfd. Additionally, since this "pipe" is a Unix domain socket, descriptors can be passed across it.

Figure 3 shows the current state of the server. The client is blocked in its call to open.

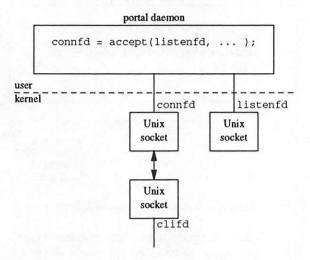


Figure 3: Client calls open, kernel creates two sockets and connects them together.

4. The portal daemon calls fork, like a typical concurrent server, and the child handles the new connection request on the descriptor returned by accept. The child closes listenfd, the parent closes connfd, and the parent calls accept again, waiting for the next client to open a file whose pathname begins with the portal mount point.

From this point on we show only the child, since it is handling this client's request.

5. portal_open builds a message that is written to clifd, which the child reads on connfd. The message contains the client's credentials (a portal_cred structure, shown below) followed by the remainder of the pathname that was saved in the vnode by portal_lookup (the string tcp/foo.com/smtp in our example). The child can validate the client using this information.

The structure written by the kernel is:

```
struct portal_cred {
  int pcr_flag;    /* file open mode */
  uid_t pcr_uid;    /* effective uid */
  short pcr_ngroups; /* # of group IDs */
  gid_t pcr_groups[NGROUPS]; /* gids */
    /* effective gid = pcr_groups[0] */
};
```

Figure 4 shows the current state of the server child.

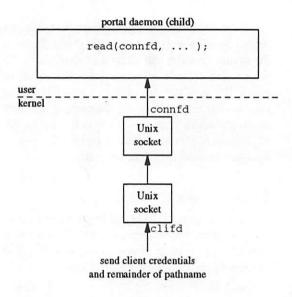


Figure 4: Kernel sends client credentials and remainder of pathname to child.

6. The child reads the portal_cred structure and the remainder of the pathname, and does whatever it needs to do with the client request. Notice that the client's request is implicitly specified by the pathname that the client opens. All the client has done is call open. The client is not aware of what's going on in the kernel, or that the portal daemon and its child are involved.

Continuing our example, the child creates a TCP socket and connects it to the specified server on the specified host. Assuming this succeeds, we have the sockets shown in Figure 5.

7. The child calls sendmsg on connfd to pass two items back to the kernel: a 4-byte error code and optionally the descriptor tcpfd that becomes the descriptor returned to the client by its open. The descriptor is returned only if the error code is 0.

The child passes the file descriptor to the kernel using the descriptor-passing capabilities of sendmsg. The descriptor is received by the

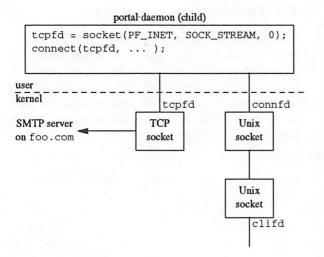


Figure 5: Child creates TCP socket and connects to specified server.

portal_open code in the kernel using the kernel equivalent of recvmsg on clifd. The child closes tcpfd, closes connfd, and terminates.

If a nonzero error code is returned by the child, that value is returned in the client's errno and the client's call to open returns -1. In our example the error code could be ECONNREFUSED (connection refused), EHOSTUNREACH (host unreachable), or ETIMEDOUT (connection timed out). These error codes extend the ones listed on the open(2) manual page.

Figure 6 shows the status of the various sockets when the child passes the TCP socket to the kernel.

8. If a descriptor is passed by the child (the error code is 0), it is converted into the lowest unused descriptor in the client process and becomes the return value from open. We call this tcpfdl. Its integer value will probably differ from the value of tcpfd in the child, but both of these descriptors refer to the same file table entry within the kernel. This is a basic property of passing descriptors: the descriptor value probably differs between the sender and the receiver, but both descriptors refer to the same file table entry in the kernel. Figure 7 shows the status of the TCP socket in the client, after the client's call to open returns.

The 4.4BSD implementation of portals requires about 1,000 lines of C code within the kernel. The

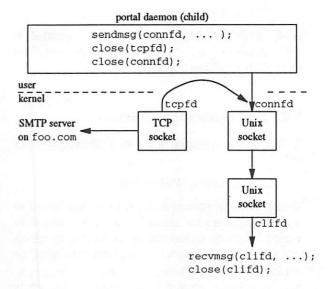


Figure 6: Status of descriptors while tcpfd is passed from child to kernel to client.

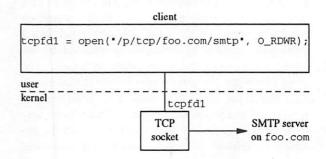


Figure 7: Final status of TCP descriptor.

portal daemon is another 1,200 lines of C code. Additional objects added to the portal namespace increase the size of only the portal daemon. The size of the kernel routines does not change.

7. Examples

7.1. TCP Namespace: Clients

The 4.4BSD implementation of the portal daemon creates TCP client connections when the pathname consists of

tcp/host/service

where *host* is either a domain name (searched for by gethostbyname) or an IP address, and *service* is either a service name (found in /etc/services) or a decimal port number. For example

tcp/news.host.com/nntp

Additionally the string /priv can be appended to the pathname to specify that a privileged port (less than 1024) should be bound to the socket.

7.2. TCP Namespace: Servers

A TCP server is created when the pathname consists of

tcplisten/IPaddr/service

IPaddr is the IP address of a local interface bound to the server's listening socket. The kernel will only accept incoming connection requests that arrive on this interface. The special value of 0.0.0.0 (called the wildcard) allows the server to accept incoming connection requests on any local interface. service is either a service name or port number that specifies the server's well-known port.

There are two problems with this type of portal service. First, the portal daemon child calls listen and accept, waiting for the client connection request to arrive. If all is OK, the connected socket descriptor returned by accept is returned by the child and becomes the return value of open. The child closes the listening socket before it terminates. This means the server that calls open is an iterative server, and not a concurrent server. There is no way to leave the daemon child in existence, with the listening descriptor still open, waiting for additional client connections.

The TCP server could fork its own child to handle the new client, after open returns, and then call open again, to wait for the next client connection. But this leaves a window of time when a listening server does not exist on the server's well-known port. Any client connections arriving in this window will be actively rejected by TCP.

Second, servers normally need to know the identity of the client. A server written using sockets would call getpeername to obtain the client's IP address and port number once the connection had been accepted. The server can use this information to validate the client. With portals, the daemon child could call getpeername after accept returns, but there is no way to return this to the server as another return value from open. Unless the server is written using a language that provides access to the getpeername function, the client's identity is unavailable to the server.

7.3. An Echo Server and Client

[Stevens 1990] shows the C code for a TCP echo server and client. The client reads a line from standard input, sends the line to the server, reads back the echoed line from the server, and writes the echoed line to standard output. Each requires about 60 lines of C code. Here is an equivalent server in Tcl [Ousterhout 1994] using portals.

```
#!/usr/contrib/bin/tclsh
set f [open */p/tcplisten/0.0.0.0/7654* r+]
while {[gets $f line] >= 0} {
    puts $f $line
    flush $f
}
```

The port number of 7654 is the value chosen for this client and server.

Here is the corresponding client in Tcl.

```
#!/usr/contrib/bin/tclsh
set f [open */p/tcp/bsdi.tuc.noao.edu/7654* r+]
while {[gets *stdin* line] >= 0} {
   puts $f $line
   flush $f

   gets $f reply
   puts $reply
}
```

Here is the same client, written using the Korn-Shell.

Using command-line arguments, the name of the server host and the port number could be specified at run-time, instead of being hard-coded within the various scripts.

7.4. Filesystem Namespace

When the pathname begins with fs/ the portal daemon opens the named file, starting at the daemon's root. For example, an open of

/p/fs/tmp/temp.1

opens the file /tmp/temp.1, relative to the daemon's root. Since the file open mode (the second argument to open) is passed in the portal_cred structure, the daemon can open the file using the options specified by the caller.

There are two potential uses for the filesystem namespace. First, it can provide a controlled escape from a chrooted environment. Second, the portal daemon can provide access control lists (ACLs, [Curry 1992]) to allow specific users access to files to which they normally do not have access.

8. Problem Areas

8.1. Standard I/O Problems With Network Connections

In the Tcl client and server just presented, an explicit flush is performed after each write to the TCP socket. This is required since languages such as Tcl and Awk normally use the standard I/O library. On the first I/O operation after a standard I/O stream is opened, a call to the isatty function is issued by standard I/O. If the standard I/O stream (not to be confused with the streams I/O system within the kernel) is a terminal device, the stream is line buffered, otherwise the stream is fully buffered. Since a socket is not a terminal device, the stream is fully buffered. The standard I/O library will not call write until either the buffer is full (normally the buffer size is 8192 bytes) or the application calls fflush.

This is a problem because when the line of output going to the server is buffered by the client, the server will be blocked reading from its socket and the client will be blocked reading from its socket. This is called *deadlock*. This is the same problem described in Section 14.4 of [Stevens 1992] with regard to coprocesses and the standard I/O library.

8.2. Awk Problems With Network Connections

Two problems arise when using portals with the Awk language. First is the standard I/O flush problem mentioned above, because unlike Tcl, there is no way to flush an Awk I/O stream explicitly.

Second is the inability to specify the open mode of a file with Awk. (The r+ in the Tcl scripts says to open the file for read and write.) Awk opens a file read-only or write-only: there is no way to open a file for reading and writing, which is required for a full-duplex socket.

8.3. Interruptibility of Network Connections

When a process creates a TCP connection, the process is often put to sleep by the connect function, which can take seconds to complete. If the connection cannot be established, and is not actively rejected by the other end point, the call to connect normally blocks for 75 seconds. When we run a program such as a Telnet client or an FTP client, we just type our interrupt key to abort the connection attempt.

Since an open of a portal object is a single system call, and involves another process (the portal daemon), the call to open is not interruptible in the 4.4BSD implementation of portals. We can type our interrupt key as often as we like, but until the daemon returns an error after 75 seconds, the call to open cannot be interrupted. This can be frustrating to users.

9. Performance

9.1. Portals Versus Other Forms of Descriptor Passing

We ran a set of timing tests to compare the performance of portals versus other forms of client-server descriptor passing. The goal is to verify that portals do not perform worse than other forms of descriptor passing. Three tests were run.

1. Portals.

The client program opens the pathname /p/tcp/127.0.0.1/13, the daytime server on the local host. It then reads from the descriptor that is returned, writing the result to standard output (which is redirected to /dev/null). The descriptor is closed. This loop is executed 20 times.

The portal daemon was modified for this test, removing the fork of a child process. This is because the next two tests do not involve a fork, and it also removes the time required for the fork from the overall timing. Also, a dotted-decimal IP address and decimal port number are specified, to remove two more variables from the timings: the calls to gethostbyname (which involves a name server) and getservbyname. Finally, the daytime service is provided internally by the inetd server and does not involve a fork or an exec.

	descriptor passing (Sec. 9.1)			portal/direct open (Sec. 9.2)	
	BSD/386		SVR4	BSD/386	
	portals	Unix domain sockets	connld	portals	direct client
client real time	0.53	2.21	2.76	1.90	1.21
client user CPU time	0.00	0.00	0.05	0.01	0.05
client system CPU time	0.15	0.36	0.41	0.19	0.23
client total CPU time	0.15	0.36	0.46	0.20	0.28
server total CPU time	0.11	0.20	0.01	1.04	AN IN
client + server total CPU time	0.26	0.56	0.47	1.24	0.28

Figure 8: Timing comparisons: descriptor passing and portal versus direct open.

2. Berkeley Unix domain sockets.

This test is a slight modification of the client-server functions in Section 15.5.2 of [Stevens 1992]. A server is started that creates a Unix domain socket and binds its well-known pathname to the socket. The client creates a Unix domain socket and connects to the server. The client writes a one-line request to the server, specifying the name of a file to open and waits for the server to return either a descriptor or an error code. The server was modified to always create a TCP connection to 127.0.0.1, port 13. The client reads from the returned descriptor, writing the result to standard output (redirected again to /dev/null). The client closes the connection to the server. This loop is executed 20 times.

The client and server do not maintain the connection between them each time around the loop, to allow comparison with the portal version. Naturally, if the client and server maintained this connection, the continual passing of a client request followed by the return of a descriptor by the server would be faster than creating and then tearing down the connection for each request.

3. SVR4 connld streams module.

This test is similar to the previous one, but it uses the client-server connection functions based on the SVR4 connld streams module from Section 15.5.1 of [Stevens 1992].

Tests 1 and 2 were run on the same hardware (a 25 Mhz 80386) with the same operating system (BSD/386 Version 1.1, with the portal code from 4.4BSD-Lite added by the authors). Test 3 was run on the same hardware, but a different operating system (SVR4 Version 2.0). This adds some variability to the timings because the creation of a TCP connection to the local daytime server uses different TCP/IP implementations on the two different operating systems. Also, on this hardware BSD/386 provides higher resolution timing by reading the high

resolution hardware clock, while SVR4 provides timings only in multiples of the system software clock (10 ms per clock tick).

The first three columns of Figure 8 shows the timing results. All times are in seconds.

The first three rows give the real time (i.e., the wall clock time on an otherwise idle system), user CPU time, and system CPU time for the three clients. The server total CPU time was obtained by running the ps command before and after the client was run.

These numbers show that a Unix domain socket is about the same as the SVR4 connld module (which isn't surprising, since they do similar things) but the portal implementation is faster still.

9.2. Portals Versus Direct Client Open

Another set of timing tests were run to compare a client using portals versus a client calling socket and connect directly. We expect the latter to be faster, since the overhead of invoking the portal daemon and the passing and receiving of a descriptor is avoided.

1. Portal client.

The client program opens the pathname /p/tcp/sun.tuc.noao.edu/daytime, the daytime server on a different host on the local subnet. A hostname and service name are used this time, instead of the numeric values from the previous set of timing tests, to include the typical overhead involved in contacting a name server and reading the /etc/services file. The host running the name server is another host on the local subnet. Additionally, the fork of a child was put back into the portal daemon, since this is its normal mode of operation. The client reads from the descriptor that is returned, writing the result to standard output (which is redirected to /dev/null). The descriptor is closed and the loop is executed 20 times.

2. Direct client open.

The client calls gethostbyname, getservbyname, socket, and connect, the same steps performed by the portal daemon. The client reads from the socket, writing the result to standard output (redirected to /dev/null again). The socket descriptor is closed and the loop is executed 20 times.

The final two columns of Figure 8 show the timing results. All times are in seconds.

The real time for this portal version (1.90) is higher than the real time for the previous portal version (0.53). We expect this since a fork is occurring, and the child is calling gethostbyname and getservbyname before it can return the connected descriptor.

The server CPU time in these examples was obtained by running the ps -S command before and after the client was run. The -S flag causes the CPU time output to include the CPU time for all terminated children. We need to account for the CPU time of the 20 children, since that is where the calls to gethostbyname, getservbyname, socket, and connect take place.

The end result of these comparisons is that the portal client requires about 50% more clock time and about 4.5 times the CPU time as a direct client. Remember, however, that this test only measures the times to establish a connection with a server on the local subnet, receive about 30 bytes of data from the server, and terminate the connection. Most TCP connections are dominated by data transfer, not connection setup and teardown. Once the connection is established, the data transfer time is unaffected by whether the socket descriptor was obtained directly by the client, or from a portal daemon.

10. Summary

Portals can provide access to a variety of objects, such as devices, special file types, and network connections, to naive applications. This is done by placing a minimal amount of code within the kernel to detect special pathnames that are being opened, and pass these to a user-mode daemon. This daemon can do whatever it wants to provide access to the specified object. A descriptor is passed from the daemon to the kernel and back to the process as the return value from open.

Although descriptor passing and connection servers have been possible for years within major Unix systems, the use of these facilities has never been transparent. Portals are an attempt to make these facilities transparent, although portals are not without their own unique set of problems.

Portals have been implemented under 4.4BSD but could easily be implemented in any version of Unix that supports different types of filesystems.

Acknowledgments

One of the authors (jsp) is responsible for the design and implementation of portals in 4.4BSD and should be credited with the ideas herein. The other author (wrs) discovered them in the 4.4BSD release and felt they were an important concept to document for others to learn about. Our thanks to Gary Wright and the anonymous referees for their comments on this paper.

References

- Aho, A. V., Kernighan, B. W., and Weinberger, P. J., The AWK Programming Language, Addison-Wesley, Reading, Mass. (1988).
- AT&T, UNIX Research System Programmer's Manual, Tenth Edition, Volume I, Saunders College Publishing, Fort Worth, Tex. (1990).
- Bershad, B. N., and Pinkerton, C. B., "Watchdogs— Extending the UNIX File System," *Computing Systems*, 1, 2, pp. 169-188 (1988).
- Curry, D. A., UNIX System Security: A Guide for Users and System Administrators, Addison-Wesley, Reading, Mass. (1992).
- Joy, W. N., Cooper, E., Fabry, R. S., Leffler, S. J., McKusick, M. K., and Mosher, D., "4.2BSD System Manual," UNIX Programmer's Manual, 4.2 Berkeley Software Distribution, Volume 2C, Computer Systems Research Group, Univ. of California, Berkeley, Calif. (1983).
- Ousterhout, J. K., *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, Mass. (1994).
- Presotto, D. L., and Ritchie, D. M., "Interprocess Communication in the Eighth Edition UNIX System," *Proceedings of the 1985 Summer USENIX Conference*, Portland, Oreg. (1985).
- Presotto, D. L., and Ritchie, D. M., "Interprocess Communication in the Ninth Edition UNIX System," *Software Practice & Experience*, 20, S1, pp. S1/3-S1/17 (1990).
- Presotto, D., and Winterbottom, P., "The Organization of Networks in Plan 9," *Proceedings of the 1993 Winter USENIX Conference*, pp. 271-280, San Diego, Calif. (1993).

- Stevens, W. R., *UNIX Network Programming*, Prentice-Hall, Englewood Cliffs, N.J. (1990).
- Stevens, W. R., Advanced Programming in the UNIX Environment, Addison-Wesley, Reading, Mass. (1992).
- Wall, L., and Schwartz, R. L., Programming perl, O'Reilly & Associates, Sebastopol, Calif. (1991).
- Welch, B., "A Comparison of Three Distributed File System Architectures: Vnode, Sprite, and Plan 9," *Computing Systems*, 7, 2, pp. 175-199 (1994).
- Welch, B. B., and Ousterhout, J. K., "Pseudo Devices: User-Level Extensions to the Sprite File System," *Proceedings of the 1988 Summer USENIX Conference*, pp. 37-49, San Francisco, Calif. (1988).

W. Richard Stevens is the author of UNIX Network Programming (Prentice-Hall, 1990), Advanced Programming in the UNIX Environment (Addison-Wesley, 1992), TCP/IP Illustrated, Volume 1: The Protocols (Addison-Wesley, 1994), and coauthor with Gary R. Wright of TCP/IP Illustrated, Volume 2: The Implementation (Addison-Wesley, 1995). He prides himself on having no degrees in computer science, instead obtaining a B.S.E. in Aerospace Engineering from the Univ. of Michigan (1973) and a Ph.D. in Systems Engineering from the Univ. of Arizona (1982). He can be reached via email as <rstevens@noao.edu>.

Jan-Simon Pendry is a graduate of Imperial College, London where he did postgraduate research on environments to support logic engineering. He is probably best known as the author of *Amd*, the freely available and widely ported automounter, and also as the author of six of the pseudo-filesystems in 4.4BSD. Jan-Simon has worked for IBM and CNS and is now a consultant with Sequent Corporation in Europe. You can contact him via email at <jsp@sequent.com>.

Dynamic Vnodes - Design and Implementation

Aju John Digital Equipment Corporation

Abstract

Dynamic vnodes make the UNIX kernel responsive to a varying demand for vnodes, without a need to rebuild the kernel. It also optimizes the usage of memory by deallocating excess vnodes. This paper describes the design and implementation of dynamic vnodes in DEC OSF/1 V3.0. The focus is on the vnode deallocation logic in a Symmetric Multi-Processing environment.

Deallocation of vnodes differs from the familiar concept of dynamically allocated data structures in the following ways: the legacy name-cache design implicitly assumes that vnodes are never deallocated, and the vnode free-list needs to cache unused vnodes effectively.

1. Introduction

Digital's DEC OSF/1 uses the Virtual File System (VFS) [OSF93, Langerman90] to support multiple file-system types. VFS is a subsystem that provides a uniform means of accessing the files in the system. Vnodes [Kleinman86] are an integral part of the VFS layer and are used to store the file index information structure of the underlying file-system, such as a UFS inode [Bach86, LoVerso91]. A vnode table is a static array of vnode structures that is generally pre-allocated at boot time. The dynamic vnodes implementation was part of a larger scalability project whose objective was to eliminate the static file-system tables in DEC OSF/1.

Dynamic vnodes free the system from the limits imposed by the static vnode table and prevent memory from being taken up by excess vnodes. Making a table dynamic is a familiar process. It involves allocating each element when there is a demand and, when it is no longer in use, deallocating it to reclaim resources. However, in the case of vnodes, the task is more complex because it is necessary to:

- ensure that a vnode will not be referenced by the name-cache code after it has been deallocated
- strike a balance between caching vnodes on the free-list and deallocating excess vnodes, depending on the system's current demand for vnodes

The discussion in this paper covers the design and implementation of dynamic vnodes. The allocation policy is a basic enhancement over the design used in the Open Software Foundation's reference version 1.1.3 of OSF/1 source code. However the reference version did not support vnode deallocation, so it was not possible to reclaim the memory used by vnodes. The implementation of dynamic vnodes resolves all the challenges associated with deallocating vnodes, making it possible to grow and shrink the number of vnodes based on demand.

In addition, the implementation ensures that the deallocation of vnodes does not impact the overall performance of the system. This is done by making the implementation sensitive to demand for vnodes. The implementation also improves the chances of reusing useful file system data that are cached in the free vnodes on the free-list.

This paper will describe the design and show its integration into the existing vnode management structure. The design is easy to integrate into virtually any kernel that uses 4.4BSD-style or OSF/1-style VFS, that is, a kernel that allows vnodes from one file-system type to be re-allocated [OSF93] to another type. The design uses a timestamp based technique to deal with the legacy name-cache design. This technique can be applied in general to deal with legacy data structures that may have references to deallocated memory.

2. Motivation for dynamic vnodes

The need for a kernel that can scale its responses to the demand for vnodes was the motivating factor for this design and implementation. In short, the kernel should be able to dynamically respond to varying demand for vnodes, without a need to be rebuilt when it runs out of vnodes. At the same time, it should release memory taken up by excess free vnodes. From a performance perspective, the deallocation design should be able to deal with references to deallocated vnodes in the name-cache table, without having to search all name-cache entries. In addition, the deallocation policy has to be sensitive to the system's demand for vnodes. It should retain an optimal

number of vnodes on the free-list for effective caching and balance it with the amount of memory consumed by vnodes.

2.1 Limitations of a static vnode table

A kernel based on a static vnode table can handle demand for vnodes up to a maximum determined by the size of the table. Such a kernel would have to be re-configured and re-built whenever the working set of vnodes exceeds the table size. Several versions of UNIX in the market, including DEC OSF/1 Version 2.0 and earlier, use a static vnode table. Its size was determined at startup, usually based on the value of maximum number of users that is set for the system. Vnodes were assigned from this table when needed and returned to be re-used or reclaimed when they were no longer needed.

When processes on such systems request an additional vnode over the table limit, the processes are suspended indefinitely or exit with an error message such as "Vnode table full". This problem became increasingly common in environments that had a large number of users.

In such a situation, the solution was to rebuild the kernel with a larger value for the maximum number of users, which in turn would create a larger vnode table, and then reboot with the new kernel. On the other hand, such a demand for vnodes could only be for a short duration. So, a kernel with a huge static vnode table might be able to handle most types of vnode demands when they happen, but this would waste memory resources during periods of low vnode demand. Consider this example, where the current size of a vnode in DEC OSF/1 V2.0 is 488 bytes: A kernel that has a vnode table that holds 20,000 vnodes will take up about 10MB of memory, regardless of the number of vnodes used.

The Open Software Foundation partially addressed this problem in the OSF/1 reference code by replacing the static vnode table and allocating vnodes when new ones are needed. This eliminated the problem of the system running out of vnodes. However, in the OSF/1 reference code, once a vnode is allocated, it is not deallocated. The Open Software Foundation probably decided to defer implementing the vnode deallocation in OSF/1 reference code due to the inherent design of the vnode name-cache, which implicitly assumes that vnodes are never deallocated.

2.2 Problems with allocation-only policy

Earlier internal releases of DEC OSF/1 had the vnode allocation-only scheme, with the added func-

tionality to cache vnodes for a certain duration on the free-list. If the vnode at the head of the free-list is fairly new, a new vnode is allocated when needed, instead of recycling the one at the head of the free-list. Soon, it was noticed that a significant amount of memory was being dedicated to the ever increasing number of vnodes and the buffers associated with the vnodes.

DEC OSF/1 tries to cache the vnodes on the free-list for a while, so that any of them can be quickly re-activated if a vnode lookup operation succeeds. The advantage of caching vnodes on the free-list for at least a fixed duration is that it prevents the cached vnodes from being flushed by simple operation such as *ls -lR(1)* [DEC-RP]. The LRU (Least Recently Used) nature of the free-list combined with the caching of free vnodes for a fixed duration causes the free-list size to grow during such operations. This allows the frequently used vnodes to get re-activated instead of re-used, while the LRU nature of the free-list pushes the other vnodes to the head of the free-list for re-use. Re-activating more vnodes from the free-list is a definite performance win.

However, if vnodes cannot be deallocated, operations such as *ls -lR* can use up a significant amount of memory in the form of newly allocated vnodes. A sudden, but not too uncommon spike of vnode requests caused by programs such as a *find(1)* [DEC-RP] operation can also take up a significant amount of memory for vnodes.

Consider a mid-sized machine such as the *Digital AlphaServer 2100 4/200*. Under low vnode demands, the system typically uses about a 1000 vnodes. However, it can allocate up to a maximum of 41,000 vnodes (Section 4.1) that takes up 5% of the 384MB of memory in a typical configuration. Over 19MB of memory will be taken up by vnodes if the system is unable to deallocate excess vnodes.

As the preceding example illustrates, there was a pressing requirement to design and implement truly dynamic vnodes that could be allocated and deallocated based on demand. The deallocation policy had to be robust enough to deallocate excess free vnodes, while maintaining an effective cache of frequently used vnodes on the free-list.

2.3 The name-cache design issue

VFS uses the name-cache or the Directory Name Lookup Cache (DNLC) [Langerman90] to improve the pathname translation performance by providing fast lookups of frequently used file names. The name-cache poses the biggest challenge to vnode deallocation. The name-cache is indexed by a hash

value on the pair *vp*, *name*, where *vp* is the vnode that refers to the directory that contains *name*. Once a reference to a vnode gets cached in the name-cache, a lookup operation on that vnode can happen any time in the future. If the lookup operation picks up the reference to the vnode after it has been deallocated, the kernel will crash trying to access invalid memory. When memory allocated to a vnode is deallocated, all cache references to that vnode must be removed from the name-cache.

The process of ensuring that all references to a vnode are removed is a challenging one for many reasons. Several different entries in the name-cache may refer to the same vnode. The name-cache is not indexed just by the vnode alone and the mapping between names and vnodes is a many-to-one type. It is prohibitively expensive to walk the entire name-cache, looking for every vnode that is targeted for deallocation, to ensure that all cache references to that vnode have been removed. Re-designing the name-cache to maintain a linked list of all the entries that correspond to a vnode reduces performance in a multiprocessing environment (Section 4.3.2) and introduces race conditions.

Therefore, a low-overhead time-stamp based technique was designed to deal with references to deallocated vnodes in the name-cache table.

3. Dynamic vnodes

Vnodes that can be allocated on demand and deallocated to return the resources used by them are termed *dynamic vnodes*.

Dynamic vnode design discussed here can be used to modify any kernel that has the 4.4BSD-style or the OSF/1-style VFS layer. The chief characteristic of this type of VFS layer is that vnodes are not bound to any one file-system type. In such an implementation, a vnode can get recycled from one type to another after it has been cleaned. Before cleaning it, the *vgone()* operation eliminates all activity associated with the vnode. The *vclean()* routine cleans the vnode by doing the file-system specific reclaim operation on it. This operation disassociates the file-system dependent data from the vnode before re-use. Other implementations of VFS, such as the one in SunOS, are different from the type described here.

3.1 Design Principles

Dynamic vnodes were designed so that the following conditions were always adhered to:

- · dynamically configurable limits
- low overhead
- re-use all of the existing VFS logic and code
- multi-processor safe

Dynamically configurable limits allow any limits in the design to be changed by the administrator on a running kernel, totally transparent to all other users in the multi-user mode.

Low overhead deallocation was a necessity to ensure that the overall performance of a kernel with dynamic vnodes would be as close as possible to the kernel with static vnodes or one with dynamic allocation alone. It was also necessary to handle name-cache entries of vnodes that were deallocated, without incurring the high cost of search operations.

The emphasis was to adhere to the original VFS code and to integrate the deallocation code into the existing code base. This has two advantages. It will still make use of all the time-tested VFS code that was already in place and, if needed, the new code can be easily turned off or excluded, making the system just as it was before the change. In fact, one of the design criteria was to make it possible to switch the kernel from a deallocation mode to a no-deallocation mode at run time. This was essential to understand and improve the performance of the deallocation code.

The new design had to be safe for use in a symmetric multiprocessing environment, without any race conditions between the deallocation and the lookup code paths. At the same time, the design had to avoid being riddled with locks, which would reduce performance.

4. Design and Implementation

Dynamic vnode design is discussed in two parts. Allocation of vnodes is the first part, which is an enhancement to the design from the Open Software Foundation's reference version 1.1.3 of OSF/1 source code. This will be briefly discussed with emphasis on the enhancements. The second part is the vnode deallocation design, which will be discussed in detail.

4.1 Allocation of vnodes

The allocation scheme uses the *power-of-two* [McKusick88] allocator to allocate vnodes when needed. At system start-up, there are no vnodes in the system. The first vnode is dynamically allocated in

the VFS initialization routine. A new function called *initnewvnode()* allocates a new vnode and returns its address after initializing all of the fields. The *to-tal_vnodes* count is incremented with each vnode that gets allocated.

To make optimum use of memory while using the power of two allocator, the vnode structure was re-sized to be under 512 bytes. This was done by rearranging the fields so that the structure was packed tight and also by eliminating flag fields and lock fields that could be merged with other existing fields. In order to keep the code intact, new macros were written to make the vnode structure changes transparent to the rest of the VFS code that uses it.

The following terms are important to this discussion:

- max-vnodes: the maximum number of vnodes that can be present in the system.
- minimum-free-vnodes: the minimum number of free vnodes that should be present in the system for vnode caching.
- vnode-age: a value in seconds that is used to determine if a vnode at the head of the free-list can be recycled rather than allocating a new vnode.

Although in theory vnodes can be allocated without an upper bound, a soft limit called *max-vnodes* was introduced to prevent runaway allocations. The value of *max-vnodes* is set when the system boots, depending on the amount of memory in the system. By default, the *max-vnodes* value is set to the number of vnodes that can fit in five percent of the system memory.

The VFS getnewvnode() function allocates a new vnode by calling initnewvnode() if the number of free vnodes is less than minimum-free-vnodes or if the vnode at the head of the free-list is not older than vnode-age. This gives the vnode at the head of the free-list a chance to be reclaimed and makes vnode caching more effective. Thus, on a busy system, the vnode free-list tends to be longer. A new vnode is always allocated if the number of free vnodes is less than minimum-free-vnodes.

The values of max-vnodes, minimum-free-vnodes, and vnode-age can be changed dynamically on a running system using a kernel debugger or by using the sysconfig(8) [DEC-RP, DEC-ST] utility.

4.2 Deallocation of vnodes

When the number of free vnodes in a system is fairly high and the rate of usage of vnodes in the system is fairly low, vnode deallocation routines re-

lease the memory held by the excess vnodes back to the system.

4.2.1 Name-cache complications

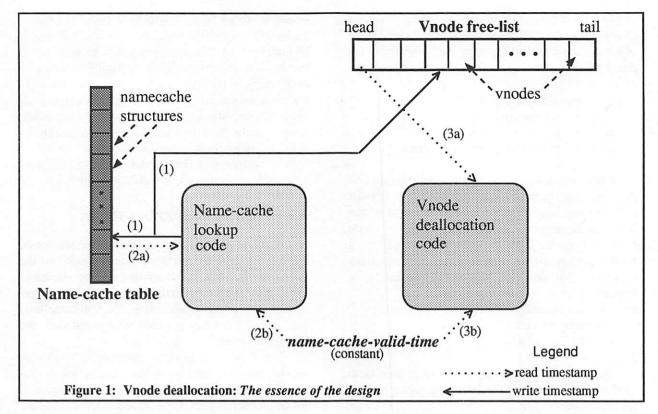
With most system data structures, deallocation would have been as simple as freeing the memory assigned to the vnodes in an LRU fashion. Because the vnode free-list is maintained in an LRU order, vnode deallocation would appear as a simple task of freeing vnodes closer to the head of the list.

However, there are references to vnodes in the name-cache that are used for quick pathname translation [OSF93, Langerman90]. The design of the namecache assumed that references to vnodes are always valid and did not take into account the fact that memory allocated to a vnode can get freed. It depended on the vnode identifier field in the vnode called v id to determine whether the reference to the vnode is valid. When a vnode is deallocated, it is not economical to search the name-cache for obsolete entries to get rid of them. This is because several different name-cache entries may refer to a single vnode and the number total number of entries in the name-cache can be very large. On the other hand, when a lookup operation returns a reference to a vnode, that vnode should not be a deallocated vnode. A simple cache-purge operation alone would not suffice because the method it uses to invalidate a reference to a vnode is to increment the vnode's generation number. If the vnode is going to be deallocated, the contents of that vnode will be lost and incrementing the generation number will not work.

The problem is compounded by the fact that the logic had to be safe for a multiprocessing environment. When deallocating, races had to be avoided without getting into a deadlock [Bach84]. The following races are possible if vnode deallocation is introduced:

- a vnode can get deallocated between a vnodelookup operation and a vnode-get vget() operation.
- a vnode-lookup operation can race with the deallocation operation on a particular vnode.
- name-cache purge operations for vnodes due to umount operations can race with the deallocation operations for the vnodes.
- vital counters can get corrupted if allocation races with deallocation, resulting in system errors or memory leaks due to lost vnodes.

Our solution implements vnode deallocation by using time-stamps based on the value of *lbolt*.



4.2.2 Timestamps using Ibolt

The kernel variable called *lbolt* stores the number of hard-clock interrupts since boot. Using a very reliable timestamp mechanism was critical to the design. The system-time was not used for timestamps because it can be changed on a running system. The main reason for using *lbolt* is that its value is guaranteed to increment, and never to decrement. It is also easy to convert the lbolt value into seconds by a low-overhead shift operation based on the system's frequency.

4.2.3 Design of the deallocation logic

The deallocation code introduces the following kernel variable, which plays a key role in the design:

 name-cache-valid-time: name-cache entries with timestamps older than this value are considered stale and are discarded. Vnodes that have timestamps older than this value are chosen for deallocation.

Vnodes from the head of the free-list that have not had any name-cache activity in the last *name-cache-valid-time* seconds are considered for deallocation. During a lookup operation, if the name-cache entry is older than *name-cache-valid-time*, the associated vnode may have been deallocated, so its address is not returned. The value of this variable can

be set at boot-time, but cannot be tuned on a running kernel.

Figure 1 illustrates the working of the logic. Timestamp fields are present in the vnode and the name-cache structures. In the figure, the timestamp update process is represented by solid lines, while timestamp read operations are represented by dotted lines. The name-cache lookup code is invoked whenever there is a need to add or to look up an entry in the name-cache table. When an entry gets added, that entry and the corresponding vnode are time-stamped (1). When a sought entry is found in the table during a lookup operation, its time-stamp is checked (2a) to see if the entry is older than name-cache-valid-time (2b). The lookup operation returns the entry if it is not older than name-cache-valid-time and the timestamps of the entry and the corresponding vnode are updated (1) atomically. If the name-cache entry is older than name-cache-valid-time, it is purged and the lookup operation returns a failure because the corresponding vnode could have been deallocated.

Figure 1 also illustrates the vnode deallocation code that inspects the timestamp on the vnode (3a) at the head of the free-list that is being considered for deallocation. It is compared (3b) with the value of name-cache-valid-time. One of the conditions necessary to deallocate a vnode is to have its timestamp older than name-cache-valid-time.

Clearly, the vnode deallocation code depends on timestamps as a part of its design. New fields were added to the vnode and the name-cache structures for this purpose.

```
struct vnode {
   /* new fields only */
   u_int v_ncache_time;
   u_int v_free_time;
   u_int v_cache_lookup_refs;
```

Every vnode has two new timestamp fields, one to store the time it was put on the free-list (v_free_time) and the other to store the time of the last name-cache activity for that vnode (v_ncache_time). The v_cache_lookup_refs is a counter that keeps track of pending vget operations on vnodes that have been successfully looked up in the name-cache.

```
struct namecache {
   /* new field only */
   u_int nc_time;
}
```

Every name-cache entry has a new timestamp field (nc_time) to store the last name-cache activity for that entry.

4.2.3.1 Resolving the name-cache issue

When name-cache activity such as a cache-lookup or cache-enter operation happens, the name-cache-activity field in the vnode (v_ncache_time) and the nc_time field in the name-cache entry are updated simultaneously with the current time index based on the value of lbolt. Whenever a vnode is put on the free-list, the current lbolt-timestamp value is stored in the vnode's v_free_time field.

The free-list time field (v_free_time) is used by getnewvnode() to determine if a vnode at the head of the free-list needs to be cached for a longer time or if it can be recycled. The last name-cache activity timestamp field on a vnode (v_ncache_time) is used by the deallocation code to determine if that vnode can be deallocated. If its value is older than name-cache-valid-time, that vnode is a candidate for deallocation.

Thus, the deallocation code guarantees only to deallocate a vnode if there was no name-cache activity for that vnode for a period of time determined by the value of *name-cache-valid-time*. The name-cache code depends on this guarantee made by the deallocation code to decide whether to send back references to vnodes for successful cache-lookup operations. If the name-cache activity timestamp on a cache entry that was successfully looked up is older than the predetermined *name-cache-valid-time*, the

vnode reference is not returned. Instead, that name-cache entry is deleted and a counter for such hits is incremented. In this way, references to deallocated vnodes can be avoided without having to examine every entry in the name-cache for possible deletion. By counting the number of bad cache hits due to an old timestamp, the loss in performance of the name-cache can be determined. This helped in tuning the value of *name-cache-valid-time* to make the cache-lookup performance loss under 0.1% for different types of load conditions, as shown in Section 5.

4.2.4 When to deallocate a vnode:

A good time to check for excess free vnodes on the free-list is when vnodes are being added to the tail of the free-list. If the number of free vnodes in the system is greater than the value of *minimum-free-vnodes*, the vnode deallocation function *vdealloc()* gets called whenever a vnode is released onto the free-list by *vrele()*.

The *vdealloc()* function attempts to deallocate vnodes from the head of the free-list, which is protected by a free-list lock. For every vnode selected for deallocation, its *v_cache_lookup_refs* field is examined to see if there are any *vget()* operations pending after a successful cache-lookup operations for that vnode. If so, this vnode can get referenced any time and is not a good choice for deallocation. Next, the timestamp field *v_ncache_time* for the last namecache activity is examined. If any cache-lookup activity has happened in the last *name-cache-valid-time* seconds, the deallocation code must honor the guarantee and return the vnode to the head of the free-list.

To reduce the number of times the free-list-lock gets called, it tries to deallocate vnodes in groups. To defray the cost of deallocation across several *vrele()* operations, only a fraction of the excess vnodes are deallocated by each *vdealloc()* call.

Every *getnewvnode()* operation also inspects the vnode it takes off the free-list and deallocates it if it meets the deallocation criteria. To prevent *getnewvnode()* from getting too busy deallocating old vnodes, a configurable number sets a limit on the number of vnodes that it can deallocate during each call.

4.2.5 Tuning deallocation to system load.

A special vnode called *marker-vnode* is injected into the free-list chain. The special vnode is never removed, but it cycles constantly through the vnode

free list from the tail to the head. When it reaches the head, the <code>getnewvnode()</code> function detects it and puts it at the tail of the free-list, where it will continue traveling toward the head of the free-list. Each time it is put at the end of the free-list, it is timestamped. By examining the marker vnode, the deallocation code can determine the cycle-rate of free vnodes. If the vnodes in the free-list are cycling at a rapid rate, the deallocation code will back off until the vnode activity on the free-list is slow. When the cycle-rate is not too rapid, the deallocation code can consider deallocating vnodes from the head of the free-list.

Vnode deallocation is temporarily turned off for a tunable period of time if any of the following conditions are met:

- deallocation attempts fail more than a predetermined number of times per batch attempt.
 Deallocation will not happen unless either a predetermined number of vnodes have been removed from the head of the free-list for re-use or a pre-determined amount of time has elapsed.
- the number of free-vnodes falls below the value of minimum-free-vnodes.
- the marker-vnode is cycling through the free-list at a rate faster than once per vnode-age seconds.

4.2.6 Handling SMP Races

This section describes how the races encountered in a Symmetric Multi-Processing (SMP) environment were handled by the design.

4.2.6.1 Handling the vget() race

The v cache lookup refs field in the vnode is used to keep track of any successful cache-lookup references for that vnode. Normally, a successful cache-lookup operation is followed by a vget() operation to remove the vnode from the free-list. In between the cache-lookup and the vget() operations, the vnode should not get deallocated. So, if there has been a successful cache-lookup operation, the v cache lookup refs field value is incremented, which serves as a hint to the deallocation code not to deallocate that vnode. This prevents the race between vnode deallocation and vget(). Later, when the vget() operation happens on the vnode, the counter is decremented. In the kernel, the vget() function calls after cache-lookup operations are replaced by a macro called vget cache(), which causes the lookup count to be decremented. reference The v cache lookup refs field in the vnode is also protected by the same global simple-lock used to protect

the timestamp fields. For some reason, if any code in the kernel decides not to do a vget() operation on a vnode after a successful cache-lookup operation, it should call a function called abort-vget(), which just decrements the cache-lookup-reference count for that vnode. If this is not done, the vnode will not be deallocated. The $v_cache_lookup_refs$ field is also incremented by iget() when it finds a vnode it needs on the free-list using vget(). This prevents the vnode from being deallocated while iget() is holding a reference to it.

4.2.6.2 Handling the cache-lookup race

A possible race condition exists where a vnode can get deallocated while it is being looked up in the name-cache. To prevent this from happening, the simple-lock that is used to protect v cache lookup refs field in the vnode has to be acquired by the deallocation operation and the cachelookup operation. This prevents the cache-lookup operation and the vnode deallocation operation from racing with each other. If the cache-lookup operation succeeds in acquiring the lock first, it increments v_cache lookup refs field in the vnode as discussed earlier (Section 4.2.6.1). If deallocation code succeeds in acquiring the simple-lock, the cache-lookup operation will have to wait until the lock is released by the deallocation code. The subsequent timestamp check in the cache-lookup operation will ensure that the vnode will not be referenced if it has been deallocated.

4.3 Other Avenues explored

During the process of design and implementation, several variations of the design were considered and a few of them were prototyped. This section discusses some of them and reasons why they were not used.

4.3.1 Name-cache references in the vnode.

An earlier implementation of the design did not use the *name-cache-valid-time* method. Instead, every time a name-cache entry was made for a vnode, the address of the name-cache entry was stored inside the vnode structure. Vnodes were deallocated based on the age of the vnode on the free-list. When the vnode was being deallocated, it had a reference to the name-cache entry corresponding to it, which could be purged at the same time. This design had a major drawback because a vnode could have multiple name-cache entries, while the vnode could only store the latest entry. Multiple name-cache entries for a

vnode occur when there are multiple hard-links to a file. If a cache-lookup operation found a matching entry that was not the latest one for a vnode, it would end up referencing a vnode that was already deallocated and result in bad memory access in the kernel.

4.3.2 Shadow name-cache

This design was to take care of the fact that multiple name-cache entries were possible for a vnode. When a name-cache entry for a vnode is being made, the name-cache address field in the vnode is checked to see if the vnode has an earlier entry. If so, the first and the second name-cache entries are linked by adding an extra address field to the name-cache entry structure. In this way, if a vnode had multiple name-cache entries, every entry will have a shadow entry link to the previous entry until the link reaches the first entry for the vnode. When the time comes to deallocate a vnode, the link from the first entry down to the last entry is followed and deleted.

This approach works very well on uni-processor systems, but has race problems in multi-processor systems. Several locks had to be introduced to follow the links to all the name-cache entries of a vnode. Each step required holding and releasing locks, and then looking back to see if anything had changed in the environment. The increase in the use of locks has a negative impact on the performance on multi-processor platforms. Despite these efforts, the implementation of this design was hard hit by race conditions.

4.3.3 Attaching vdealloc to the idle thread

Instead of having the vnode deallocation code attached to *vrele()* and *getnewvnode()*, it was considered a better idea to have a separate thread execute the code, thus allowing the task to be done when the system was relatively idle. So, the initial plan was to attach this code to the idle thread in the kernel. Further study showed that this was not feasible because the vnode deallocation code could block trying to get a vnode off the vnode free-list, as the free-list lock could be held by other processors in an SMP environment. A future enhancement of the design would be to use a separate kernel thread to execute this code.

5. Performance

Dynamic vnodes introduces a certain amount of overhead, but the design criteria was to have the per-

formance very close to that of a similar system without dynamic vnode deallocation. It was also the goal to closely monitor the name-cache performance and to ensure that it is close to the case where namecache entries are not invalidated based on their age.

Initially, a number of small exerciser programs were used to specifically test the vnode allocation/deallocation code. These programs forced the system to allocate a large number of vnodes and later release them to the free-list. These were originally designed to exercise the new code path but were also used to compare the performance of the systems.

Later, to find the overall performance of the system, standard benchmarks were used and the results were compared with the base-line reference provided by the version of the operating system before the introduction of vnode deallocation. All along, the name-cache performance was monitored by periodically extracting the name-cache statistics, called nchstats. A new counter called bad_time_hits was introduced to keep track of all matches that were thrown away as a result of the age of the entry being older than name-cache-valid-time.

In all cases, the cache-performance results showed that the discarded hits were less than 0.1% of the total good hits.

5.1 AIM Performance

5.1.1 Description of the test

AIM Suite III benchmark was developed by AIM Technology to test the total system performance of all major system components in a multitasking environment [AIM93]. Suite III attempts to simulate the load that a specified number of users would exert on a computer system by running a set of functional benchmarks intended to model a particular application. The set of functional benchmarks is scaled by the number of simulated users to arrive at the desired load. Besides file-system exercisers, these functional benchmarks also consist of memory, floating point, pipe, and CPU exercisers. This test was used to compare the performance of the kernel with and without vnode deallocation to determine if the overhead of having vnode deallocation was significant. The test results reported here are based on the Standard Model, which purports to describe a typical UNIX environment.

5.1.2 Setup for the test

The test was performed on a DEC 7000 AXP system running DEC OSF/1 Version 3.0 (Revision 344) with the following configuration:

- four DECchip 21064/182MHz
- 512MB RAM
- Prestoserve NVRAM
- six RZ28 (4GB) disks, two controllers

The tests were done with the network interfaces (Ethernet/FDDI) turned off and the test system was isolated from any external stimulus.

5.1.3 Results

The AIM Suite III Systems Throughput Comparison graph (Figure 2) shows the throughput comparison of two tests. The first test was conducted with vnode deallocation enabled (default). In the second test, vnode deallocation was turned off to avoid the overhead introduced by the deallocation code. It is evident by comparing the results of the two runs that there is hardly any drop in overall system performance when deallocation of vnodes is enabled. Note that the difference between the two curves is not al-

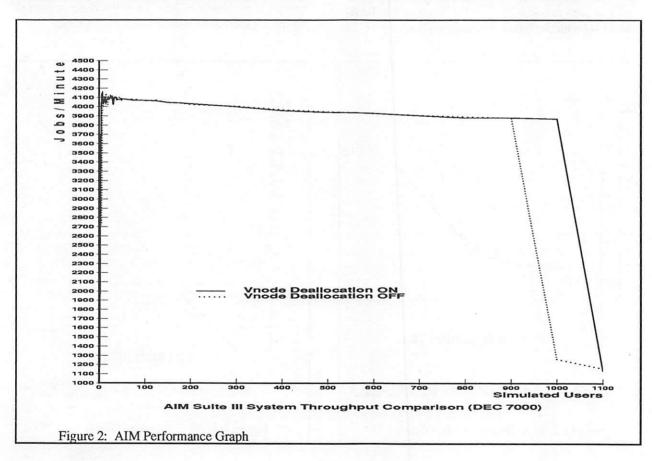
ways observed when the load exceeds 900 users. Therefore it cannot be attributed to the vnode deallocation code.

The name-cache usage statistics shows the performance of the name-cache. The statistics gathered after the AIM test from 1 to 1100 users are:

bad_time_hits	28
ncs_goodhits	51397595
ncs_neghits	75276
ncs_badhits	833068
vn_allocations	18299
vn deallocatio	ns 10939

From the statistics, 28 good hits had to be rejected from a total of 51,397,595 due to the possibility of the corresponding vnodes being deallocated. This works out to be less than 0.000054% when the name-cache-valid-time value was set at 20 minutes. This shows that the two main design requirements were met in the implementation:

- overall performance remains unchanged after the implementation of dynamic vnodes.
- the name-cache performance drop is well under 0.1%, even though the name-cache entries older



than name-cache-valid-time were being discarded.

5.2 UIW Performance

UIW is an acronym for a benchmark called ULTRIX Integrated Workload. This is used to determine the performance of a release of the operating system by measuring its response-time and throughput for varying load conditions.

5.2.1 Description of the test

UIW is a workload that attempts to simulate the activities of various members of a software development project on a time-sharing system. The workload is composed of a mix of commands and utilities that are file and CPU intensive (ls, cat, grep, make, cc, awk, etc.) [DEC-RP] running in a canned environment. The selection and frequency of execution of the processes are based on the observations of the activities of a software development unit in Bell Laboratories.

5.2.2 Setup for the test

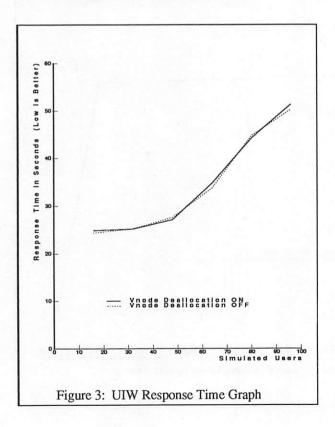
The test was performed on a DEC 3000-500 AXP system running DEC OSF/1 Version 3.0 (Revision 344) with the following configuration:

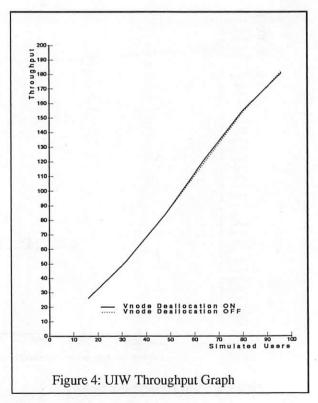
- one DECchip 21064/150 MHz (uni-processor)
- 256 MB RAM
- four RZ28 (4GB) disks on a single controller

5.2.3 Results

From the results shown in Figure 3, the response time for the system that deallocates vnodes is close to that of the system that does not deallocate vnodes. In this graph, a system with lower response time is considered better. Figure 4 shows that throughput changes with increasing number of users are similar. Throughput is measured as the number of times the commands were executed in a duration of three hours, for a given number of simulated users. At the end of the run, the cache statistics were examined for losses due to vnode deallocation. From a total of 23,228,647 good hits, 9,425 hits were rejected because the name-cache entry was older than namecache-valid-time. The loss was about 0.0405% of the good hits, which is well below the design requirement of 0.1%. Each UIW test takes about 48 hours to complete.

The results from the UIW tests show that the overall performance of DEC OSF/1 was not impacted by the addition of the vnode deallocation logic.





6 Impact of the design

6.1 Impact on tools

As a result of the introduction of dynamic vnodes, all application and debugging programs that depended on the static vnode table locations had to be modified. They can no longer obtain the starting address of the vnode table, the size of the vnode table, and the offset to the start of a particular vnode. Instead they can only access the vnodes from the various lists by walking down the pointers in these lists. These lists will include the vnode free list and all of the mount lists. The mount lists are connected to each other in a circular list, starting with the *rootfs*. All kernel debugging tools that access vnodes were modified in DEC OSF/1 V3.0 to gather information on vnodes from the mount-lists and the free-lists.

6.2 Impact of tunable parameters

The tunable value of *name-cache-valid-time* can impact the performance of the *namecache*. If the value is set low, the deallocation will happen at a faster rate, while the number of bad cache hits will also increase. Higher values of *name-cache-valid-time* will slow the rate of deallocation, but will decrease the number of bad cache hits.

The tunable value of *vnode-age* will impact the size of the vnode free list. Large values will cache free vnodes for a longer time before the vnodes are re-cycled, but will increase the length of the free-list. If its value is set low, the vnodes will be re-cycled faster and allocation will be less aggressive, but the number of free vnodes cached will be less.

6.2.1 Selecting the default values

The design goal was to keep the name-cache performance loss due to cache invalidation under 0.1%. The value of *name-cache-valid-time* has a direct impact on name-cache performance. Starting with a value of 10 minutes, the value was incremented based on the feedback obtained from name-cache performance statistics recorded in the *nchstats* global kernel structure. A value of 20 minutes was chosen since it kept the losses under 0.1% for different types of loads. At the same time, it was important not to delay the deallocation process by choosing a very large value. A large value for *name-cache-valid-time* will weaken the deallocation process, leaving a large number of free vnodes in the system.

It is hard to come up with a default value for *vnode-age* that would be ideal for all types of work loads. In *vnode_stats* kernel structure, the *vn_vgetfree* field indicates the number of vnodes that were successfully taken off the vnode free list. If this value is too low, the value of *vnode-age* may be raised to increase the minimum time a vnode will remain on the free-list. A default value of 2 minutes is used. This value can be fine tuned on a running system, depending on the type of jobs that typically run on it. Its value will require further fine tuning in the future, depending on the type of machine and the type of load.

The value of *max-vnodes* is made sensitive to the amount of memory in the system. It's default value is calculated at boot time to the number of vnodes that can fit into 5% of memory. Although this value may seem high, only a fraction of that memory is used most of the time because it is just the upper bound for dynamic allocation of vnodes. Its value can be changed at any time on a running kernel.

The default value of *minimum-free-vnodes* is directly proportional to the configured value of the maximum number of users for the system. If necessary, this can also be changed on a running system. A kernel that has been configured for 32 users will typically have a default *minimum-free-vnodes* value of 468 vnodes.

7. Related Work

The ULTRIX Engineering Group at Digital developed a prototype kernel based on ULTRIX V2.0, where many of the static tables were replaced by linked lists [Rodriguez88]. However, vnode deallocation was not implemented.

The Open Software Foundation had done part of the work where vnodes get allocated, but did not implement the vnode deallocation code.

Similarly, IRIX Version 5.0 from SGI allocates vnodes and maintains a global free-vnode pool from which vnodes can get re-assigned.

4.4BSD-Lite (April 1994) also allocates vnodes, but does not deallocate them. According to sources from University of California, Berkeley, 4.4BSD-Lite is more advanced than the 4.4BSD-Encumbered, and vnode deallocation is not currently done in either of them.

8. Future work

Several areas can be worked on in the future to make the deallocation code even more optimal. Some of the areas are:

- adding a dedicated thread to do the deallocation of vnodes, instead of attaching it to vrele() and getnewvnode() code paths. This will reduce the extra overhead placed on the two calls.
- increasing lock optimizations and improving lock granularity of the locks used.
- doing name-cache entry invalidation based on name-cache activity time and the age of the vnode that was most recently deallocated, instead of on name-cache-valid-time.
- fine-tuning the default values of the tuning parameters to make deallocation even more optimal. Fine tuning is needed in the case of vnodeage to make vnode caching on the free-list more optimal.
- checking the amount of data associated with the vnode object before deallocating a vnode. It may be far less expensive to deallocate a vnode that has fewer pages of data associated with it.
- converting the static name-cache table with a dynamically managed list based on the number of vnodes in the system. The static name-cache table has been temporarily retained during the first phase of implementation, as the name-cache structures are a lot smaller than vnode structures.

9. Conclusions

Although the use of dynamic data structures is a well-known concept, the dynamic vnodes design shows new techniques to counter some of the issues due to the inherent nature of the existing code:

- data structures that contain references to a structure that has been deallocated can easily be purged by using timestamps. This eliminates the need for expensive searches to purge such data structures. This also eliminates the need to keep track of all possible references to a structure that can get deallocated.
- caching of free structures is more effective when the free structures are retained for a set duration, provided that there is enough memory. This bal-

- ances the cache efficiency with memory consumption and prevents inadvertent cache flushes.
- tracking a marker element in a LRU list helps as an indicator of the traffic on the list. This information can be used to make the design sensitive to different patterns of demand.

These are general purpose techniques that can be used to resolve issues that are similar in nature to those encountered by dynamic vnodes.

Dynamic vnodes provide the system with all of the advantages of dynamic data structures. Because their allocation and deallocation is based on demand, they make the system more responsive to varying loads and increases its capacity to meet varying demand for vnodes. In this design, vnodes do not take up any more memory than required because none of the vnodes are pre-allocated. Dynamic vnode implementation in DEC OSF/1 also provides improved vnode caching on the free-list, thereby reducing disk access.

The results from the benchmarks show that the implementation of the deallocation design does not add any measurable overhead. The percentage of good name-cache hits lost due to aging is within tolerance limits.

9.1 Availability

Dynamic vnodes with allocation and deallocation are a part of DEC OSF/1 Version 3.0, which was released in August 1994.

10. Acknowledgments

I would like to thank all the members of the file-system group for their support. Special thanks to Stan Luke who motivated and supported me during the design and implementation. Paul Shaughnessy, Chet Juszczak, Eric Werme, and Jim Woodward helped me at various stages of this work. Diane Lebel provided invaluable technical reviews of various drafts of this paper. Bob Hapgood and Marty Lund helped with the performance measurements. Ken Hall improved the readability of this paper. Rich Draves at Microsoft, my USENIX shepherd for this paper, provided valuable suggestions and detailed reviews beyond the call of duty. Cheryl Wiecek, Rich Cascio, and Andy Kegel supported me in publishing this work.

Bibliography

[AIM93] "UNIX System Price Performance Guide", Summer 1993, AIM Technology, Santa Clara, CA.

[Bach84] M. J. Bach, S. J. Buroff, "Multiprocessor UNIX Operating Systems", *AT&T Bell Labs Technical Journal*, Vol 63, October 1984, 1733-1749.

[Bach86] M. J.Bach,"The Design of the UNIX Operating System", *Prentice-Hall, Inc.*, 1986, Chapter 4.

[DEC-RP] "DEC OSF/1 Reference Pages, Version 3.0." Digital Equipment Corporation, Maynard, MA.

[DEC-ST] "DEC OSF/1 System Tuning and Performance Management, Version 3.0." *Digital Equipment Corporation, Maynard, MA.*, 1994, Chapter 3.

[Kleinman86] S. R. Kleinman. "Vnodes: An Architecture for Multiple File System Types in Sun UNIX. *Proceedings of the USENIX Summer '86 Conference*, 238 - 247.

[Langerman90] A. Langerman, J. Boykin, S. LoVerso, S. Mangalat, "A Highly-Parallelized Machbased Vnode Filesystem", *Proceedings of the USENIX Winter* '90 Conference, 297 - 311

[LoVerso91] S. LoVerso, N. Paciorek, A. Langerman, "OSF/1 UNIX Filesystem", *Proceedings of the USENIX Winter '91 Conference*, Jan 1991, 207 - 218.

[McKusick88] M.K. McKusick, and M. J. Karels, "Design of a General Purpose Memory Allocator for the 4.3 BSD UNIX Kernel", *Proceedings of the USENIX Summer Conference*, June 1988, 295 - 303.

[OSF93] The Open Software Foundation,"Design of the OSF/1 Operating System - Release 1.2", *Prentice Hall, Inc.*, 1993, Chapter 11.

[Rodriguez88] R. Rodriguez, M. Koehler, L. Palmer, R. Palmer, "A Dynamic UNIX Operating system", *Proceedings of the USENIX Summer Conference*, June 1988, 305 - 319

Author Information

Aju John is a software engineer in the UNIX Engineering Group at Digital, where he works in the file systems area of DEC OSF/1. He has worked at Digital since 1991. Aju got his M.S. in C.S. at the Worcester Polytechnic Institute in Worcester, Massachusetts in 1991. Reach him electronically at aju@zk3.dec.com or via U.S. Mail at Digital Equipment Corporation, ZK03-3/U14, 110 Spit Brook Rd, Nashua, NH 03062-2698.

All trademarks and registered trademarks are the property of their respective holders.

Union Mounts in 4.4BSD-Lite

Jan-Simon Pendry
Sequent UK

Marshall Kirk McKusick Author and Consultant

ABSTRACT

This paper describes the design and rationale behind union mounts, a new filesystem-namespace management tool available in 4.4BSD-Lite. Unlike a traditional mount that hides the contents of the directory on which it is placed, a union mount presents a view of a merger of the two directories. Although only the filesystem at the top of the union stack can be modified, the union filesystem gives the appearance of allowing anything to be deleted or modified. Files in the lower layer may be deleted with whiteout in the top layer. Files to be modified are automatically copied to the top layer.

This new functionality makes possible several new applications including the ability to apply patches to a CD-ROM and eliminate symbolic links generated by an automounter. Also possible is the provision of per-user views of the filesystem, allowing private views of a shared work area, or local builds from a centrally shared read-only source tree.

1. Background

Sun's Translucent Filesystem (TFS) [Hen90a] provides a method for viewing the contents of a list of directory trees as if they were unioned together. The desire was to provide a filesystem-level interface to the problem of source code control. TFS follows on from work done at Bell Labs such as the 3-D Filesystem [Kor89a] by providing automatic data copying from one directory to another as new source versions are created and edited. TFS was originally implemented entirely outside the kernel but poor performance led to the addition of special support code inside the kernel.

The Plan 9 implementation of *union mounts* is fundamental to the way that system mounts are implemented. A *before*, *after*, or *replace* option can be given to the mount system call. The union operation only takes place at the mount directory itself as compared to the union filesystem that merges the namespace all the way down the directory tree. Similar semantics were chosen for an earlier version of the union filesystem but proved too restrictive for general purpose use in 4.4BSD.

The Spring operating system [Kha93a, Mit94a] makes use of a fully object-oriented implementation

technique along with a well-defined set of object methods. Spring supports an implementation of stackable filesystems without requiring special support within the filesystem code itself, purely by taking advantage of the object invocation mechanism and the callbacks that can be used to provide a cache coherency protocol. While these mechanisms appear superior to the vnode stacking architecture in 4.4BSD-Lite, they do not, by themselves, provide any of the namespace and copy-up functionality of the union filesystem.

Other systems such as DOS and VM/CMS provide a way of *appending* a directory to an existing directory such that the contents of both are visible at a single place. Both systems limit appending to a single level of namespace.

2. Introduction

The 4.4BSD kernel vnode architecture is similar in spirit to the original Sun design [Kle86a]. As the demand grew for new filesystem features, it became desirable to find ways of providing them without having to modify the existing and stable filesystem code. One approach is to provide a mechanism for stacking several filesystems on top of each other [Ros90a].

The stacking ideas were refined and implemented in the 4.4BSD system [Hei94a].

The traditional filesystem mount mechanism in 4BSD provides a way to attach complete filesystems to directories in the system namespace. Namespace construction is flexible but the limitation of mounting only complete filesystems can cause some problems in disk space management.

The namespace of a system is constructed from the namespaces within each individual mounted filesystem. Each file system is mounted at a place chosen by the system administrator and many filesystems may be created to lay out the namespace in a controlled fashion. This organization can lead to a conflict between allocating the available disk space to filesystems (wanting to allocate all the disk space to a single filesystem) and the desire to build the most easily used and managed namespace (wanting to create many small filesystems). Earlier releases of BSD did not provide any relief from this conflict. The null and union filesystems allow a new approach to disk and namespace management; they allocate all the disk space to a single filesystem, then attach sub-trees of the filesystem to the namespace as required. This approach allows the disk space to be shared between uses without need of manual intervention by a system administrator. In addition, the namespace in the system can be created and managed as required by users and applications.

4.4BSD provides two filesystem namespace management mechanisms: the null filesystem and the union filesystem. The null filesystem provides a simple flat namespace attachment mechanism, and the union filesystem provides more dynamic hierarchical functionality. These two mechanisms and some possible applications are described below.

3. Null mount filesystem

The *null mount* filesystem is a stackable filesystem in 4.4BSD that allows arbitrary parts of the filesystem tree to be mounted anywhere else in the system namespace. This functionality can be used to glue together several directories to form a new directory tree. One use of this functionality would be to rearrange filesystem trees on multiple disks into one managed tree that is presented to users. Alternatively, the new mount point could be the same as the original directory. This approach allows a sub-tree of a writable filesystem to be made read-only.

The null filesystem implements namespace attachment by attaching the original directory beneath the new mount point. With one exception, all filesystem

operations are intercepted and directed to the original filesystem. The exception is the fetch file attribute operation (VOP_GETATTR) that modifies the returned data to reflect the namespace of the new mount point (in particular, ensuring that *getcwd*(3) and *df*(1) function correctly).

The null filesystem continues to obey the traditional semantics whereby a mounted filesystem hides the files already present in the directory on which it has been mounted.

4. Union mount filesystem

The union mount filesystem is a conceptual extension of the null filesystem. The difference is that the union filesystem does not hide the files in the mounted on directory. Instead, it merges the two directories (and their trees) into a single view. The union filesystem allows multiple directory trees to be simultaneously accessible from the same mount point. This functionality is similar to that described in [Kor89a]. Sun's TFS uses an entirely different mechanism to provide some of the union filesystem functionality.

At mount time, directories are *layered* above or below the existing view. This structure is called a *union* stack.

4.1. Union mount semantics

A mount stack has a physical and a logical ordering. The physical ordering is the temporal order in which directories were mounted. The last directory mounted in a stack must be the first to be unmounted. Restricting unmounts to the last mounted directory is because of historical Unix semantics that have been carried over into the 4.4BSD-Lite VFS/vnode data structures. If this restriction were to be relaxed then a mechanism for naming lower-layer filesystems would be required. One possibility would be to use an *escape* from the namespace such as the 3-D filesystem's '...' which provides access the next lowest layer [Kor89a].

The logical ordering is that seen by filesystem namespace operations. Directories can be mounted either logically at the top or the bottom of the existing view. The logical ordering will be the same as the physical ordering whenever directories are added from above. It will differ from the physical ordering when a directory is logically mounted below the existing view. When accessed through the mount stack, all but the topmost logical layer are read-only.

A union stack where the logical and physical ordering is identical can be thought of as a sequence of lazily evaluated cp -rp commands.

4.2. Namespace operations

The union filesystem operates on the filesystem namespace. Special actions are required whenever a request is made to change the namespace of the lower layer, for example using commands like mv, rm or chmod. Since the lower layer is read-only, the union filesystem can only make changes in the upper layer. This restriction leads to several special cases:

Copyup operations

Whenever file contents or attributes are changed the target file is copied to the upper layer and changes are made there. To maintain the lazy copy semantics, the new copy of the file will be owned by the user who did the original mount, not the user who triggered the *copy-up* operation. In addition, the umask at the time of the mount is applied, not the umask at the time of the copy-up. Operations that cause a copy-up include link, chmod, chown and open for write. See section 4.5.

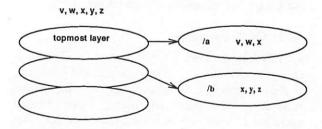
Copyup operates on only one name at a time. Other links to the same file are not copied and so the link count in the upper layer will be incorrect.

Whiteout operations

If a name is being removed from the lower layer's namespace a *whiteout* is created in the upper layer. A whiteout (see below) has the effect of masking out the name in the lower layer. Operations that cause a whiteout to be created include unlink, rmdir and rename.

4.3. Union naming

A directory listing of a mount stack shows all the files in all the directories involved in the stack (figure 1).



Union stack on /mnt (Logical ordering)

Figure 1: Stack view

In this example, /b is layered over an empty directory /mnt, then /a is layered over the new view of /mnt.

Duplicate names are suppressed so that only one occurrence of \mathbf{x} (and also \mathbf{x} and \mathbf{x}) will appear.

A name lookup will locate the logically topmost object with that name. The lookup progresses down the mount stack in the logical stacking order doing a lookup at each layer and generating a list of vnodes corresponding to each layer. In figure 2 the name 'x' will be found in the layer mounted from /a whereas the name 'y' will be found in the layer mounted from /b. The upper or lower layer vnodes could themselves be union vnodes whenever more than two layers were in the stack or if two union stacks were joined.

Two special cases occur during lookup.

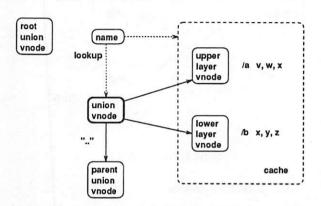


Figure 2: namespace

4.3.1. Lookup for . .

A lookup for '...' is treated as a special case. The need for this special case is illustrated in figure 3. The current directory is /tmp/fred, a directory that exists in the upper layer, but does not have a corresponding directory in the lower layer. When looking up '..', the process wants to reference /tmp which does have corresponding directories in both layers. If the union vnode for the /tmp directory had fallen out of the cache, then it would need to be reconstructed. To do the reconstruction would require finding the upper and lower level vnodes that the union vnode must reference. The /tmp directory for the upper layer could easily be found by looking up the '..', entry for /tmp/fred, but there is no way to find the /tmp directory for the lower layer since there is no directory in which to lookup a "..." entry. To avoid this dilemma, each union directory vnode maintains a reference to its parent ('parent union vnode' in figure 3).

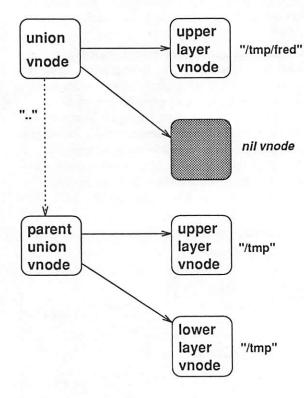


Figure 3: looking up . .

4.3.2. Directory lookup

When a directory is looked up in a lower layer and there is no object with the same name in the upper layer, the union filesystem automatically creates the corresponding upper layer directory, known as a shadow directory. These semantics have the side-effect of populating the upper layer directory tree as the union stack is traversed (a find will cause it to be fully populated). The upper layer directory is created to ensure that a directory exists should a copy-up operation be required. The copy-up needs a directory in which to create the upper layer file.

The shadow directories could be created at the time of the lookup or at the time of the copy-up. By creating shadow directories aggressively during lookup the union filesystem avoids having to check for and possibly create the chain of directories from the root of the mount to the point of a copy-up. Since the disk space consumed by a directory is negligible, creating directories when they were first traversed seemed like a better alternative. Within the union stack the existence of additional directories is transparent to the user.

4.4. I/O operations

Having located a name, a sequence of filesystem operations is executed. As an example, consider a read I/O request (figure 4). Here, a file with a matching name exists in the lower layer, but not in the upper layer. So, the read I/O operation will be directed to the lower layer vnode.

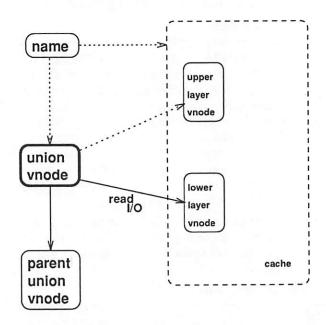


Figure 4: Read I/O

Now suppose that the file in the lower layer was opened for writing. Since the file is in the lower layer, it cannot be modified (only files in the topmost layer may be modified). Instead the union filesystem automatically copies the file to the upper layer and continues the open operation on the new copy. This operation is known as a *copy-up* (see figure 5). The underlying copy remains unchanged and is not deleted.

4.4.1. Cache coherence

4.4BSD-Lite does not provide a vnode cache coherence protocol. A cache coherence protocol is needed, at least in principle, to support stackable filesystems such as a compression filesystem. Other research operating systems such as Spring provide a more elegant and complete stacking mechanism that includes provision for full cache coherence mangement [Kha93a, Kha93b].

It was not within the scope of this work to design and implement a suitable cache coherency protocol for 4.4BSD-Lite. So, the current union filesystem implementation ignores the problem of detecting changes in

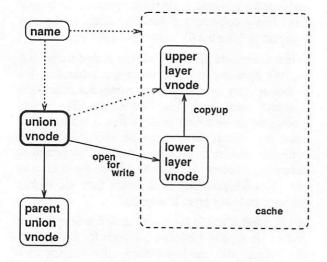


Figure 5: Copyup operation

the lower layer. The lack of cache coherency shows up when a lower layer is visible elsewhere in the filesystem and is being modified. Because shadow directories are never deleted by the union filesystem, the upper layer could retain a shadow directory whose corresponding lower layer directory had been deleted.

4.4.2. I/O performance

I/O performance through a union stack is similar to that of the underlying physical filesystem. Since no data or attributes are copied or cached within the union stack, the only performance degradation is that caused by the additional levels of functions calls through the vnode interface – one additional vnode function call for each layer in the union stack.

4.5. Access checks

The file access check made through a union stack is slightly more complex than that made for the file itself. To enforce the "lazy copy" semantics the following two tests are made:

- If a lower layer object exists then the user who did the original mount must have read permission on it.
- The current user must have the appropriate permission on the topmost object.

Test 1 ensures that access to a file or directory is not given away unwittingly by letting a user create a shadow directory in the top layer and then use the top-layer permissions. The credentials of the user who did the mount are used since the intent is to mimic a lazy copy by that user. Test 2 is the straightforward

test that the topmost object can be accessed by the user.

Suppose user floyd creates a union stack and there is a file in the lower directory also owned by floyd with mode -rw-r--r-. As expected, this file would not be writable by a different user, biema. Moreover, although the copy-up functionality exists, the new upper layer file would still be owned by floyd (as the user who did the mount) and so the permissions would still not be alterable by biema.

To be able to modify the file, biema would have to copy it to another name, remove the original file (creating a whiteout) and then move the copy back to the original name replacing the whiteout. This sequence of operations are identical to the operations required on existing filesystems and demonstrates how the semantics of file permissions are preserved even through a union stack.

5. Whiteouts

A whiteout is a directory entry that hides any identically named objects in the lower layers. Any reference to that name returns the error code ENOENT. It is possible to detect the existence of whiteouts in a directory using the '-w' option to ls(1).

5.1. Creation and deletion of whiteouts

Whiteouts are created automatically by the union filesystem whenever a namespace modifying operation is executed that cannot be implemented by normal changes solely in the topmost layer. For example, attempting to remove a file in a lower layer results in a new whiteout being created in the upper layer, Whiteouts are removed automatically when a new file or directory with a matching name is created. They can also be explicitly removed using the '-W' option to the rm(1) command, causing the lower layer file to reappear.

Suppose an *unlink* is executed on the name 'x' shown in figure 1. When the name in the /a layer is removed the object in the /b layer will become visible. To prevent the lower layer object from showing through, a whiteout for the name 'x' is created in the topmost layer. Similarly, if an unlink is executed on the name 'y' a corresponding whiteout will be created. However, no whiteout would be created if the file 'w' was removed since there is no visible object with the same name in any of the lower layers.

Whenever a whiteout is replaced by a directory, the directory inode is automatically tagged with the

opaque attribute¹. During a lookup in a union stack, the union filesystem does not continue to the lower layer object if the upper layer object is marked opaque. To understand this requirement, consider the sequence of operations:

rm -rf tree
mkdir tree

After the rm command has completed, there will be a single whiteout in the upper layer covering the name tree. When the mkdir command is run, the whiteout is replaced with a directory in the upper layer called tree. Without the *opaque* attribute, the normal union semantics would make the lower layer tree visible once more. By marking the newly created upper layer directory with the *opaque* attribute the union filesystem ensures that the semantics prevents the lower layer objects in tree from becoming visible. Should it be desirable to 'undelete' the underlying object, the *opaque* attribute can be turned off manually using the *chflags*(1) command.

5.2. Whiteout implementation

Directory entries in 4.4BSD were extended to include a tag field that records the type of the associated file. This tag was added specifically for the benefit of functions such as the file tree walker, fts(3), which can now determine whether a name represents a directory simply by looking at the directory entry. There is no problem keeping the type field in the directory synchronized with the type on the physical filesystem since the type is set permanently at the time of creation.

The semantics of a whiteout for a given name is that a lookup or delete operation on that name should return "No such file or directory" (ENOENT).

A natural way to implement whiteouts is to put a small amount of additional logic into the kernel directory scanning function. The current implementation defines a new directory type, DT_WHT. A whiteout is created simply by adding a new directory entry with the required name, but with DT_WHT as the type. Although no physical inode is allocated, a non-zero file number must be present so that the entry will not be skipped by those programs that ignore directory entries with file number zero. Thus, whiteouts are assigned the file number wino. For UFS filesystems, wino has the numerical value one that is otherwise

unused. If a file in the upper layer is being deleted, and hence a whiteout is also being created, the filesystem simply converts the directory entry to the new type and updates the file number.

When a directory entry of this type is discovered during the directory scan, the scan stops (since it has found an entry with the correct name) but returns the ENOENT error code described above. The kernel nameidata structure, records that a whiteout has been discovered, rather than no entry whatsoever. This information is used by the union filesystem to determine whether to do a lookup in the lower layer or not. If a whiteout has been found then the lookup through the lower layer is skipped.

An alternative would have been to define a new inode and vnode type and allocate an inode for each white-out object. Had an inode been allocated for each whiteout it would be possible that an over quota error could occur while executing an unlink system call. The directory tag implementation makes running out of resources much less likely. (An out of space error could occur if the top level directory would have to expand to hold the whiteout name, and no disk space was available in the top level filesystem to fulfill the expansion request.)

The fts(3) library identifies whiteout entries and returns information about them. Programs that need to see whiteouts use the FTS_WHITEOUT flag when calling fts_open. This flag implements the ls -W and find ... -type w commands.

6. Suppression of duplicate names

Since the union filesystem makes the contents of many directories visible at one place, one issue to consider is that of duplicate names. Duplicate names occur whenever a name appears in more than a single layer in the union stack.

There are two approaches to this issue. The simplest way is to ignore it entirely rosulting in the name appearing twice in an 1s listing. This approach is taken by Plan 9. Multiple identical names cause user confusion, since the user cannot easily discover which one they will really get if they try to access it. It is also impossible to select only one of the names with a globbing expression. For these reasons, 4.4BSD-Lite (and TFS) eliminate the duplicate names.

For the union filesystem, duplicate suppression is implemented in the *opendir*(3) function. Whenever a union stack is being read, the entire directory is read and duplicates are removed. At the same time, whiteout entries are recognized and cause later entries with the same name to be suppressed. Commands that

¹ Files in 4.4BSD-Lite have a set of attribute flags associated with them. These flags can be used to specify that a file is append-only, immutable or not to be dumped during a backup. The union filesystem adds the *opaque* attribute described here.

scan a file tree function correctly and do not have to deal with the case of a name appearing twice.

The decision to implement this functionality in the C library was taken to avoid requiring the kernel to buffer arbitrary amounts of data while executing a sort and unique operation. A new interface, __opendir2, that has an extra flags parameter to specify the behavior to be taken with whiteouts is used by the fts(3) library to specify alternative semantics. The standard opendir is implemented by a call to __opendir2 specifying that whiteouts should be suppressed.

7. Union stack management

In the current implementation, it is the responsibility of the user to ensure that filesystems are layered as required. The expectation is that the logical ordering of the union stack will be unchanged for the lifetime of the usage of the filesystems. Changing the ordering of the layers will have the greatest effect on whiteouts. Since the whiteouts themselves are created within the filesystems, if the filesystems are re-ordered, the effect of the whiteouts will vary. At any time, a whiteout will hide names in lower layers, but whiteouts may only be created or deleted in the top layer.

8. NFS as an upper layer

The NFS filesystem does not implement whiteouts, so it is not generally possible to use NFS as an upper layer in a union stack. The exception is that if the union mount is made read-only (using mount -r...) then the mount is allowed since no attempt to create a whiteout will be made. It is always possible to use NFS as a lower layer. The implementation does not specifically know which filesystem will be used as the top layer. Instead, at mount time, the kernel calls the whiteout vnode operation for the upper layer filesystem and asks whether whiteouts are supported. Currently only UFS (fast filesystem, memory-based filesystem and log-structured filesystem) includes support for whiteouts.

9. Applications

9.1. Writable CD-ROMs

A problem with releasing software on a CD-ROM is the difficulty of updating it when patches arrive. The union filesystem can be used to address this problem by mounting an empty UFS directory on top of the CD-ROM mount point.

```
mount -t cd9660 -r /dev/sdla /usr/src
mkdir /usr/local/src
mount -t union -o above /usr/local/src /usr/src
```

By taking advantage of the union filesystem copy-up and whiteout functions, files on the CD-ROM can now be modified, renamed or removed as needed. Even programs such as *patch*(1) work as intended.

9.2. Source tree management

The union filesystem can be used to create a private source directory. The user creates a source directory in their own work area, then union mounts the system source directory underneath it.

```
mkdir ~/sys
mount -t union -o below /sys ~/sys
```

The view of the system source in ~/sys can now be edited without affecting other views of the master source since all changes remain in the private tree. When the changes are completed they can be merged back into the master tree in /sys.

9.3. Architecture specific builds

A common difficulty with public domain software is building it for several different architectures from a central (NFS mounted) source area. Usually the software must be reconfigured each time a different architecture is used and the source tree must be cleaned out. Other software uses a combination of different subdirectories or clone trees with symlinks.

The union filesystem provides a clean alternative. Simply attaching the source tree beneath a local build directory will cause all configuration changes and object files to be created locally without modifying the central source tree.

```
mount -t nfs fb:/usr/src /usr/src
mkdir /var/obj
mount -t union -o below /usr/src /var/obj
cd /var/obj/sbin/mount_union
make
```

The union stack limits all modifications to the local system and even hand made changes required for a 'quick and dirty' port will not be visible to other machines. The master source tree is still accessible via the path /usr/src so changes could be reintegrated to the master source if required.

10. Future work

The existing implementation is functionally complete as originally envisaged. Since starting to make use of the union filesystem some additional enhancements and uses have come to light.

10.1. Cache filesystem support

The existing functionality of the union filesystem is very close to that required by a simple cache filesystem. In particular, the namespace and copy-up operations are already implemented.

The cache protocol would be write-through, rather than copy-back. This approach simplifies cache coherence, especially when using an underlying filesystem such as NFS that does not provide any support for network-wide data coherence. The changes required to implement this idea would be in vnode functions such as VOP_WRITE and VOP_SETATTR that would need to modify the remote copy first, then apply the change locally. If the remote modification failed, the local copy would be flushed from the cache and a failure returned for the whole operation.

10.2. Automounter

A serious problem with using an automounter [Pen94a] is the way it returns symbolic links pointing from the automount name to the mounted filesystem. One solution to this problem is to add special support to the kernel such as has been done in Solaris 2 [Cal93a]. An alternative is to take advantage of the namespace operations provided by the union filesystem.

As an example, consider an automount point /home. This directory will be used as an automount point for users' home directories. The home directories themselves will be mounted under /automnt which will be loopback union mounted on /home. First the automounter is started and mounts itself on /home. Then /automnt is union mounted on top of /home. These operations give a stack consisting of /automnt on top of /home, with the automounter beneath it.

When a lookup for user /home/floyd arrives, the kernel will do a lookup in the union stack for that name.

- First it will do a lookup in the topmost layer of /home, which will search /automnt and fail with ENOENT,
- Then it will do a lookup in the lower layer of /home which has the automounter mounted on it; this lookup will cause the automounter to mount the appropriate filesystem on /automnt and then return ERESTART:

mkdir /automnt/floyd
mount -t nfs host:/wherever /automnt/floyd
[return ERESTART]

 The (internal) error code ERESTART will cause the system call to be re-executed from the top layer, and this time the lookup will find the name in /automnt.

Later lookups will find the name immediately the first time /automnt is visited. Periodically the automounter can try to unmount the filesystem:

unmount /automnt/floyd
if ok
 rmdir /automnt/floyd
endif

The automounter is now only responsible for mounting and unmounting named objects and no longer has to maintain the namespace seen at the automount point itself. This approach makes for a much smaller and simpler program. A simple performance improvement is possible. When the mount point is created by the automounter, it can be marked *opaque* before mounting the home directory. The *opaque* flag will prevent a full union stack traversal on later lookups.

11. Availability

The union filesystem code was first released in 4.4BSD-Lite. The refinement of the interface including the addition of whiteout was added following the initial release of 4.4BSD-Lite. The union filesystem implementation described in this paper did not appear until revision 2 of 4.4BSD-Lite.

The union filesystem does take advantage of the stacking and vnode operation vector expansion features found in the 4.4BSD vnode interface. While the code will not drop into systems lacking these features, it should be possible to port it to other vnode implementations. Several other changes have been made at user level, mostly in the C library. These changes should be straightforward to integrate into the shared C library on most systems.

References

Cal93a.

B. Callaghan and S. Singh, "The Autofs Automounter," *USENIX Association Conference Proceedings*, pp. 59–68 (June 1993).

Hei94a.

J. S. Heidemann and G. J. Popek, "File-system development with Stackable Layers," ACM

Transactions on Computer Systems, pp. 58-89 (February 1994).

Hen90a.

D. Hendricks, "A Filesystem for Software Development," *USENIX Association Conference Proceedings*, pp. 333–340 (June 1990).

Kha93a.

Y. A. Khalidi and M. N. Nelson, "An Implementation of UNIX on an Object-oriented Operating System," *USENIX Association Conference Proceedings*, pp. 469–479 (January 1993).

Kha93b.

Y. A. Khalidi and M. N. Nelson, "Extensible File Systems in Spring," TR-93-18, Sun Microsystems Laboratories, Inc. (September 1993).

Kle86a.

S. R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *USENIX Association Conference Proceedings*, pp. 238–247 (June 1986).

Kor89a.

D. G. Korn and E. Krell, "The 3-D File System," *USENIX Association Conference Proceedings*, pp. 147-156 (June 1989).

Mit94a.

James G. Mitchell, et al, "An Overview of the Spring System," *Proceedings of Compcon* (February 1994).

Pen94a.

J-S. Pendry and N. Williams, "Amd – The 4.4BSD Automounter Reference Manual" in 4.4BSD System Manager's Manual, pp. 13:1–57, O'Reilly & Associates, Inc., Sebastopol, CA (1994).

Ros90a.

D. Rosenthal, "Evolving the Vnode Interface," *USENIX Association Conference Proceedings*, pp. 107–118 (June 1990).

Acknowledgements

We would like to thank Keith Bostic for providing the final impetus necessary to get this work finished and published. We thank Phil Winterbottom and Rob Pike for reading and extensively commenting on a draft of this paper. While we incorporated most of their comments, we ignored their admonition that adding new options to the 1s command has been banned since 1984.

Biographies

Jan-Simon Pendry is a graduate of Imperial College, London where he did postgraduate research on environments to support logic engineering. He is probably best known as the author of *Amd*, the freely available and widely ported automounter, and also as the author of six of the pseudo-filesystems in 4.4BSD. Jan-Simon has worked for IBM and CNS and is now a consultant with Sequent Corporation in Europe. He has two cats, Floyd and Biema. You can contact him via email at <jsp@sequent.com>.

Dr. Marshall Kirk McKusick got his undergraduate degree in Electrical Engineering from Cornell University. His graduate work was done at the University of California at Berkeley, where he received Masters degrees in Computer Science and Business Administration, and a Ph.D. in the area of programming languages. While at Berkeley he implemented the 4BSD fast file system, was involved in implementing the Berkeley Pascal system, and was the Research Computer Scientist at the Berkeley Computer Systems Research Group overseeing the development and release of 4.3BSD and 4.4BSD. He is currently writing a book on 4.4BSD and doing consulting on UNIX related subjects. He is a past president of the Usenix Association, a member of the editorial board of UNIX Review Magazine, and a member of ACM and IEEE.

NOTES

MASS STORE

Session Chair: Charles J. Antonelli, CITI, University of Michigan

NOTES

Evaluation of Design Alternatives for a Cluster File System

Murthy Devarakonda, Ajay Mohindra, Jill Simoneaux,* and William H. Tetzlaff

IBM Thomas J. Watson Research Center, Hawthorne, NY.

Abstract — Based on implementation experience and measurements, this paper presents an evaluation of design alternatives to a cluster file system. The file system is targeted for IBM cluster systems, Scalable POWERparallel and AIX HACMP/6000. We considered a shared disk approach where serialized, multiple instances of a single-system file system directly access file data as disk blocks, and a shared file system approach which is the conventional method of distributing file system function between a client and a server. We conclude that the shared disk approach suffers from the difficulties of metadata serialization, poor write-sharing performance, and read throughput.

1 Introduction

Based on implementation experience and measurements, this paper presents an evaluation of design alternatives to a cluster file system. The file system is targeted for IBM clusters, Scalable POWERparallel [1] and AIX HACMP/6000 [2]. VAXClusters [3] are other well known cluster systems. In general, a cluster is a loosely coupled system of processing nodes, each with its own CPU, memory, disks, and network adapters. The cluster systems considered here also have characteristics such as single administrative domain, peer relationship among nodes, homogeneity, and special purpose hardware (i.e. high-speed switch or multiported disks). A cluster file system, unlike a conventional network file system, can take advantage of these characteristics for high performance and fault tolerance. Functionally, a cluster file system is expected to provide strong cache consistency, POSIX [4] support, and high availability.

We considered two design approaches to the cluster file system: (1) a shared disk approach, where multiple, serialized instances of a single-system file system run on the cluster nodes, directly accessing the file data as disk blocks; (2) a shared file system approach, which is the conventional client/server scheme using the vnode interface. Note that compatibility with the existing disk format is a requirement so that users can easily migrate to a cluster from a single system.

This evaluation is based on implementations in AIX/6000 using the same cache consistency mechanism and the cache manager. Cache consistency has been implemented using a distributed token manager [5], and AIX virtual memory has been used as the data cache. In the shared disk approach, the lesser known of the two approaches, the AIX Journaling File System (JFS) [6] has been extended by adding cluster-wide serialization with the help of the token manager. Remote disk access has been provided over the network. Comparison of this parallel JFS implementation with the shared file system approach is one part of the evaluation. The conclusion is that efficient metadata serialization is intrinsically difficult in the shared disk approach.

The second part of the evaluation is based on performance measurements of the two implementations. (Although adequate for an insightful evaluation, our implementation of the shared disk approach can simultaneously support only one readwrite mount and many read-only mounts.) We have used an Nhfsstone-derived micro-benchmark, Andrew benchmark based scalability measurements, read and write throughput measures, and a concurrent benchmark. Results show that the shared file system approach performs better in concurrent read-write access and in reading a large

^{*}J. Simoneaux was an employee of IBM while working on this project; She is now with the Intel Corp.

file sequentially, whereas the shared disk approach has a small advantage when almost all accesses can be serviced by the local cache. Since a file system for a homogeneous cluster can be designed using either one of the approaches, an experimental evaluation is valuable; we are unaware of such previous work.

Although fault tolerance is important in a cluster file system, we did not attempt an evaluation based on fault tolerance. However, a non-disruptive recovery scheme exploiting the multiported disk hardware has been implemented for the shared file system approach. Details can be found in a technical report [7].

The rest of the paper is organized as follows: The next section summarizes the workload expected for the cluster file system. Section 3 describes how the two design approaches relate to earlier file systems. Section 4 describes the two design approaches. Section 5 discusses serialization challenges in the shared disk approach. Section 6 presents performance measurements. Section 7 concludes the paper.

2 Workload Characterization

The application domain for VAXClusters [8] provides a working example for the types of applications suitable for clusters and cluster file systems. Based on this and our understanding of commercial applications, we anticipate two types of workloads: one, the traditional UNIX usage patterns [9, 10], and the other, transactional accesses to large files. The first type of workload is characterized by many small files, whole-file sequential accesses, mostly read accesses, little write-sharing, and high locality of reference. In a cluster, these patterns occur when the cluster file system is used to access commands, configuration files, user files, and so on. We also expect interactive computing uses of the cluster for editing, compiling, and testing of software. This usage is predominant when the users connect through Xterminals and use the cluster as a fault-tolerant, scalable computing system. For this workload, in addition to such well known techniques as caching, cache consistency, read-ahead, and write-behind, a private protocol1 on the internal high-speed switch can provide additional performance improvement.

The second type of workload is characterized by large files, random accesses to file pages, and

read-write sharing. Typically this type of workload is present when the cluster is used as a highly available transaction server. The transaction system may be a client-server application, where the server part runs on each of the cluster nodes and accepts requests from the clients running on PCs or workstations. The transaction system may be a vendor provided database or custom built application. The transaction system can be designed using a partitioned or shared data model. The partitioned model does not require a single-image file system. The application takes on complete responsibility for directing client requests to the appropriate server node and recovery from server node failures. The shared data model can exploit a single-image file system for load balanced, seamless use of the cluster. It can also make use of the fault-tolerant characteristics for high data availability.

3 Related Work

The shared file system approach has been used in many existing distributed file systems such as NFS [11], AFS [12], DFS [13], Locus file system [14], and Sprite file system [15]. Unlike these file systems, our cluster file system is intended to run in a homogeneous environment, therefore a shared disk approach is a potential alternative.

Before discussing the related work in the shared disk approach, we should note that the caching and degree of cache consistency vary significantly among above mentioned file systems. Both approaches discussed here use a distributed token manager to provide strong cache consistency which is functionally similar to DFS and Sprite.

One existing file system that employs a shared disk approach is the file system for the VAXCluster [16]. The file system is not a UNIX file system, but it supports operations similar to open, close, read, and write. The authors started with a single-process (i.e. non-reentrant) version of the file system, and developed a version that supported not only multiple processes but also the cluster. We followed a similar methodology in developing the shared disk approach from JFS, except that JFS was already capable of supporting multiple tasks. One key difference is that the VAXCluster file metadata has simpler structure and layout as compared to JFS. Another important difference is that the operation of the VAX-Cluster file system is not as closely tied to the virtual memory manager as JFS. However, JFS

¹Implementations described here do not use a private protocol at this time.

derives significant performance benefits due to its metadata design and close ties to the virtual memory manager.

Welch has compared three shared file system architectures in [17], but we are not aware of any previous work that compares the shared disk approach with the shared file system approach. Since a cluster system allows both approaches to the design of a single-image file system, an experimental evaluation of the approaches is valuable.

4 Two Design Approaches

In this Section, we briefly discuss the token manager, as it is common to both approaches. Next, we provide an overview of the two approaches. Metadata serialization in the shared disk approach is discussed separately (in Section 5.1), as it requires an in-depth treatment.

4.1 The Distributed Token Manager

Tokens convey authorization to perform certain actions on file data. For example, after acquiring a read token on the data pages of a file, a node can cache the pages and allow read operations on the data until the token is revoked. In both designs, the relevant tokens are marked busy during the execution of a vnode operation. Thus, cluster-wide locking is defined as marking a token busy, and cluster-wide unlocking is defined as marking it not busy. A token, once acquired, remains with the cluster node until it is revoked by some other node. A token can be revoked only when it is marked not busy.

A node initiates token arbitration to acquire, upgrade, or voluntarily give up a token. When a node is attempting to acquire a token, the token client contacts the token server first. The token server is the node with a physical connection to the disks containing the file system. The server always maintains the current status of a token. Therefore, when a request is received, the server grants the token if there are no conflicts. If conflicts do exist, server replies with the copyset, i.e. the list of nodes holding the tokens. In the latter case, the requesting node revokes or downgrades the tokens (from write to read) at the copyset nodes, and then acquires the token.

What tokens are used and what objects they represent are important questions. However, the answers depend on the specifics of the design. Note that the token manager operation does not

depend on what objects the tokens represent but only on the token semantics (such as the singlewriter/multiple-readers semantics).

4.2 The Shared Disk Approach

In the shared disk approach, multiple instances of the standard AIX file system (JFS), each running on a different cluster node, directly access on-disk file data as disk blocks. Each JFS instance caches disk blocks as in the stand-alone system. However, because all instances are simultaneously accessing the disk blocks, it is necessary to serialize accesses and provide data consistency. This is similar to "parallelization" for a symmetric, shared-memory multiprocessor. Therefore, we call this approach Parallel JFS (PJFS). Here, however, locking is done at a larger granularity to account for the communication overhead in a cluster.

First consider the control flow resulting from file related system calls in the stand-alone AIX. The logical file system implements system calls by invoking vnode operations of a virtual file system type. In the stand-alone AIX, the virtual file system type would be JFS and hence its vnode or vfs operations are invoked. Since JFS uses the virtual memory for caching, the virtual memory manager handles file cache misses by reading from the logical volume containing the file system. Similarly, file cache flushes are written to the logical volume. A logical volume is an abstraction of a Unix disk partition, and presents a block device interface to the virtual memory manager and JFS. This block device interface is supported through a pseudo-device driver called logical volume manager (LVM).

PJFS extends this architecture to a cluster, by implementing two additional mechanisms: A remote logical volume (RLV) pseudo-device driver to provide access to a disk from multiple cluster nodes, and a limited number of modifications to JFS to invoke the token manager functions for cluster-wide serialization. Figure 1(a) is a schematic view of this design. If disks are sufficiently multi-ported, the remote logical volume device driver is not needed. However, the present hardware supports only four ports.

The JFS implementation has a convenient 'hook' for using the cluster-wide locking scheme. At the beginning of every vnode operation, JFS locks relevant, in-core inodes using single-system semaphores, and unlocks them at the end. Each

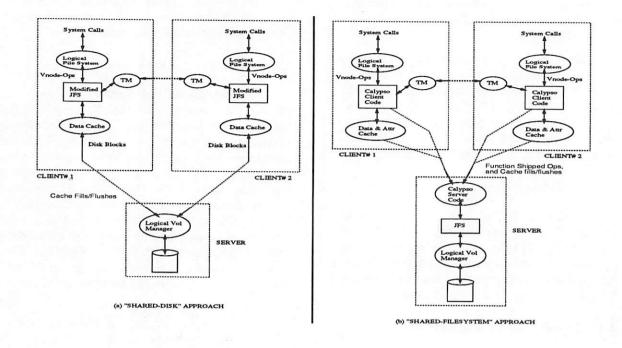


Figure 1: The Two Design Approaches

inode has a semaphore field for this purpose. We modified the inode locking function to invoke our cluster-wide locking function first and then to execute the local locking.

JFS cache data is flushed or purged as per the token protocol. When a write token is downgraded or revoked, any modifications of the file are flushed to the disks. When a token is revoked, the data and attributes of the file are purged from the cache. Existing interfaces are used to implement these flush and purge operations. After acquiring a token, caches are filled in the normal course of JFS activity.

Token Definitions In PJFS, one token represents the whole file. However, for some meta-files, which are discussed in Section 5.1, a token represents a page. For a normal file, therefore, a token represents the in-core inode, in-core indirect blocks (if any), and all the data pages of the file (Figure 2). Because JFS caches file data and indirect blocks only once, this token also represents on-disk file data and indirect blocks. However, the token does not represent the on-disk inode. JFS

caches disk inodes separately from in-core inodes,² and carefully manages the transfer of contents between disk inodes and in-core inodes. Therefore, we use separate tokens to represent disk inodes as discussed in Section 5.1.

Implementation Status of PJFS We have implemented this PJFS approach to the extent that a file system could be mounted for read-write on one node (not necessarily the node with the disk) and read-only on any number of other nodes at the same time. This was enough to carry out meaningful performance measurements as well as to develop an in-depth understanding of the implementation issues.

4.3 The Shared File System Approach

The shared file system approach is the conventional method of distributing the file system function between a client and the server using the vnode interface [18]. We call this approach Calypso. Figure 1(b) provides a schematic view of this approach. One of the cluster nodes connected

² JFS inodes have different structures on-disk and incore, however, the in-core inode is a superset of the on-disk inode.

to the disk, functions as the server for file systems on the disk. Other cluster nodes function as the clients. Similar to NFS, a node may be the server for some file systems, and a client for some others. On clients, a new virtual file system is implemented, which is shown as the Calypso client in Figure 1(b). On the server, an interface layer, shown as the Calypso server in Figure 1(b), provides interface to unmodified JFS. The clients obtain remote services from the server using Remote Procedure Calls (RPCs). In addition to the distributed token manager, this design requires the client and server parts.

Client Part The client part implements vnode and VFS operations as defined in AIX/6000. Every vnode operation first acquires cluster-wide locks on necessary files, proceeds to perform the function as required, and then releases the locks. The vnode operations make use of memory caching of file pages and attributes. File pages are cached using the virtual memory manager. Attributes (e.g., access rights, ownership, and size) are cached in inode-like structures, and name to vnode mappings are cached as in NFS. As needed, these caches are filled and flushed using remote service requests to the server. Directory changing vnode operations such as the file create are implemented entirely at the server.

Server Part The server part consists of processes and code to service remote requests from the clients. All operations use JFS vnode interface to access actual file data.

Token Definitions Although, we later evolved a more complex token definition [5], for this evaluation, we used one token to represent all accesses to the file.

5 PJFS Implementation Experience

In this section, we discuss metadata serialization and journal management for the shared disk approach. Then we present a summary of the implementation difficulties. Finally, we discuss a generalization of the implementation experience.

5.1 JFS Metadata Serialization

Similar to most UNIX file systems, JFS metadata consists of a superblock, inodes, inode extensions containing optional access control lists, indirect blocks, a disk allocation map, and an inode allocation map. However, JFS accesses

these data structures themselves as files using well known inode numbers that are inaccessible to normal users. In the next three subsections, we discuss serialization and consistency schemes for the inodes, indirect blocks, and the disk allocation map. We use these three data structures as examples to illustrate the implementation difficulties of the shared disk approach. For details on JFS, the reader is referred to Chang et al [6].

Inodes

Page-level tokens provide cluster-wide serialization of the inodes file. A read token is needed for reading an inodes-file page, and a write token is needed for writing a page. The decision to use page-level tokens is based on the observation that the inodes, as a whole, experience more sharing than a regular file, so a single lock would be a serialization bottleneck. However, locking at less than the physical page level is difficult to implement. Page-level tokens are a compromise between two extremes. As dictated by the token protocol, the inodes-file is flushed or invalidated on a page-by-page basis.

Indirect Blocks

Unlike the inodes, the indirect blocks in JFS don't have a fixed position in the indirect-blocks file. Instead, indirect blocks are mapped on demand during the use of the file system. An indirect block is a just another disk block until it is referenced. Because of this design, invalidation of indirect blocks requires unmapping and remapping of the indirect blocks when large files are write-shared.

Disk Map

The disk allocation map consists of several pages, each with control information and allocation status bits. Because of the need for journaling, JFS creates a temporary allocation map. It differs from the permanent map in allocations that have not yet been journaled. Because the temporary map can be paged out, JFS uses the same virtual memory segment as for the permanent map. As a result both are backed by the file system on the disk.

When extended to an N-way cluster system, this requires N different temporary maps (because the contents may differ), each with its own backing store. A scalable solution is to use a 'working storage' segment for each temporary map. This

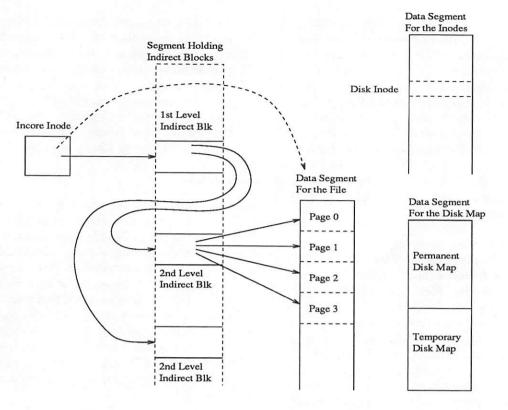


Figure 2: Components and Organization of JFS Meta-Data

type of segment can be created as needed, and backed by paging space. Note that a single, file-system backed segment (as in JFS) is not scalable. The number of temporary maps that can fit into a segment is limited.

However, using working-storage segments is not practical because it requires extensive changes to existing JFS code, which undermines our goal of keeping cluster-specific changes to a small number of lines and modules. Even if we had resolved this problem, disk map organization in units of 4K-byte pages would have caused a significant bottleneck. It would be difficult to share efficiently such a structure across the cluster nodes. Furthermore, each page contains 256-bytes of control information relating to the entire disk map.

5.2 Journal Management

In PJFS, write-ahead logging can be implemented in several ways. For example, for each file system, either a single log can be used for all cluster nodes, or a log per cluster node can be used. An intermediate solution is also possible. The single log solution has the obvious problem of

poor concurrency, whereas the multi-log solution has the following disadvantages:

- It requires a two stage recovery;
- The log records must contain a timestamp using a global clock or counter so that, at the time of merging the sublogs, the log records can be properly ordered;
- It introduces the possibility of a sublog being unavailable at the time of recovery (partial failure scenario versus total failure scenario).

Neither of the solutions seem appropriate for the purpose. So, the PJFS design is to use a single log for each file system, but instead of caching the last page of the log, the log writes are functionshipped to the server node where they are written to the log device. Usually, several log records are combined in each function shipping operation.

5.3 Implementation-Based Assessment

Meta-data serialization with high concurrency is the most difficult and is clearly the limiting factor. Specifically, the journal and inode management can be supported with a reasonable

level of concurrency, but the disk map and indirect blocks would be most difficult to support and likely to perform poorly. To achieve efficient serialization, the metadata should be designed with clusters in mind. For example, JFS uses a 4K-byte page to store the allocation information for 30, 720 disk pages, which is 120 Mbytes of disk storage. PJFS uses page-level tokens and caching for the allocation map because it is the most natural extension to JFS. However, a serialization granularity of 120 Mbytes of disk storage is too large. Therefore, the designer of a cluster file system, if starting from the scratch, would make different trade-offs regarding the metadata structures.

Another troublesome characteristic of a single-system file system is that the references to the metadata are scattered throughout the code. In a cluster system, such references must be limited to a small critical section under cluster-wide serialization.

5.4 Generalization Beyond JFS

Any existing single-system optimized file system would have the same problems as JFS. Such file systems typically cache metadata extensively using virtual memory and may even employ journaling for quick restart. Note that journaling is employed in other commercial file systems (e.g., Veritas [19] and Episode [20]) besides JFS. In clusters, distributed log management and efficient serialization of metadata are difficult. Furthermore, optimization for a single-system use is not usually the same as for a cluster.

What about non-journaling, buffer-pool based file system such as the BSD fast file system? Even in such a file system, achieving high concurrency for the metadata (such as the disk map) remains a problem because page-level granularity would still be too large. In addition, it is not clear that parallelizing the BSD fast file system would be useful, since quick restart and reliability have already been demonstrated in journaling file systems.

6 Performance Comparison

To quantify design implications of PJFS and Calypso, we used micro and macro benchmarks to measure caching efficiency, scalability, throughput, and write sharing overhead. The key findings are:

- Calypso performs significantly better than PJFS when data is write shared among the nodes;
- For cached data, PJFS has a slight edge over Calypso;
- Calypso's read throughput over the network is better than that of PJFS.

Measurements described here are taken on RISC System/6000 Model 530's on a 16 Mbit/s token ring. Each system has 80 Mbytes of memory and SCSI-2 disks. NFS (Version 2) measurements are also shown where appropriate so that the differences in PJFS and Calypso can be seen in relationship to a well known network file system.

6.1 Micro Benchmark Performance

A micro benchmark called *Fsstone* has been derived from Nhfsstone by removing NFS specific code. It measures performance of individual file operations. The benchmark opens several files in a directory, and executes a pre-defined mix of system calls such as read, write, stat, create, and unlink. The total number of system calls executed is typically large, such as 30,000, therefore the results show the positive effects of caching in the file system.

The measurements are taken with one (client) node running the benchmark on a file system residing on another (server) node. For PJFS, both the file system and the log are on the server. Results are shown in Table 1.

Table 1 shows that there is no clear winner between PJFS and Calypso. PJFS caches all types of data whereas Calypso caches most but not all. This works in favor of PJFS most of the time. However, in some complex operations, such as mkdir, rmdir, and create, caching and local manipulation of low-level metadata is expensive. Calypso's strategy of function shipping these operations is more efficient. Overall, for a representative mix of operations, PJFS has a small advantage over Calypso because such a mix is dominated by lookup, read, readdir, and fsstat operations, where PJFS has the advantage. It can be seen that both PJFS and Calypso out perform NFS due to their strong cache consistency scheme.

6.2 Scalability

Scalability measurements are another way of looking at the positive effects of caching. The

Table 1: Results from a micro benchmark showing average times for various file operations. The benchmark makes a mix of file-related system calls on a set of files.

Operation	Milliseconds/Call		
	PJFS	Calypso	NFS
get attributes	0.55	0.58	10.33
set attributes	1.59	5.47	6.81
lookup	0.69	1.03	6.12
read	0.53	0.58	4.45
write+fsync	22.75	22.25	25.94
create	24.29	21.38	45.54
remove	37.55	28.55	41.62
rename	38.95	43.18	39.22
readlink	1.09	11.75	14.06
link	39.94	34.38	44.85
symlink	51.10	35.87	39.39
mkdir	57.15	41.78	43.41
rmdir	51.89	40.69	49.05
readdir	0.93	9.53	10.41
fsstat	0.74	5.60	9.90

Andrew benchmark, which has been used in AFS scalability measurements [12], is used for the purpose. The benchmark consists of five phases, in which subdirectories are created, source files are copied, file attributes and contents are examined, and the source is compiled. Thus, this benchmark also represents the workload of interactive computing typical in research and software development.

Figure 3 shows results of the measurements. PJFS and Calypso have essentially an identical scalability characteristics as they are both based on an identical cache consistency scheme. PJFS exhibits a small edge over Calypso at each measurement point, mainly because of the performance difference in the readdir operation (see Table 1), which is prominent in the third phase of the benchmark. It can be seen that NFS performs poorly relative to both these file systems.

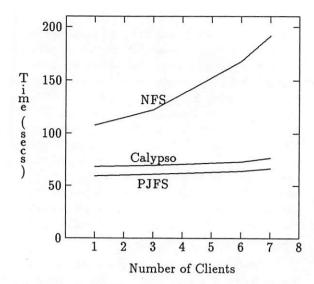


Figure 3: Scalability measurements using the Andrew benchmark. Elapsed times for the benchmark are shown for increasing number of nodes. Each client ran the benchmark in a separate directory.

6.3 Throughput Measurements

In the next two experiments, a 2 Mbyte file is written and a 2 Mbyte file is read. In the file-write experiment, the source is in the memory thus isolating the file system write throughput. Similarly, in the file-read experiment, caches are invalidated to obtain cold cache read throughput. Results of are shown in Figure 4.

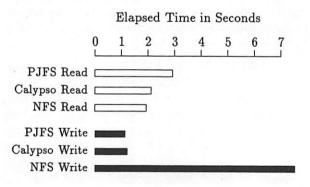


Figure 4: Throughput measurements. The first set shows read times for a 2M file under cold cache conditions. The second set shows elapsed times for copying a 2M file from memory.

Calypso and NFS show superior read throughput, as much as 30% better performance than PJFS. The reason is that Calypso and NFS use read-aheads at the client as well as at the server, whereas PJFS can perform read-aheads at the client only. On the client side, all three file systems use AIX virtual memory as the cache manager, which initiates read-aheads (by default, up to 8 pages) upon recognizing sequential access. However, on the server side, only NFS and Calypso have the advantage of substantial read-aheads as they both use local JFS to read file data from the disk. PJFS is limited to using a device driver that can perform at most one block read-ahead. (On the server, the RLV mechanism of the PJFS uses UNIX buffer management operations, bread, breada and bwrite, for disk access [21].)

Write throughput is equally good in PJFS and Calypso as they both write data to the client cache pending a flush operation at periodic or forced *sync*. NFS performance is considerably worse as it writes to server disk synchronously (particularly, when user writes as much data as in this benchmark).

6.4 Concurrent Access Performance

To understand performance trade-offs for transactional workloads where read-write sharing can be expected, we used a concurrent benchmark. The benchmark consists of reader and writer programs: The writer program repetitively writes a pre-defined number of bytes starting at the zero offset, and at the same time, the reader program repetitively reads file contents starting also at the zero offset. The elapsed times for several thousands of these simultaneous accesses are measured. To quantify sensitivity to the number of bytes accessed, we conducted two experiments, one using 8 bytes, and the other using 4K bytes. Note that AIX page size is 4K bytes. The results are shown in Figure 5.

Calypso performs significantly better than PJFS, in fact, two to five times better for this benchmark because: (1) PJFS needs to manage changes to the file and its inode separately, whereas Calypso needs to manage only the file; (2) Flushing small changes to the file system data is more expensive in PJFS than in the client code of Calypso; (3) Sharing data through a disk is more expensive than sharing through memory, even though the disk has a front-end buffer pool acting as a write-through cache.

In PJFS, to read a file, the reader first requests a token on the file. This forces the writer to flush changes using internal fsync and then yield the token. The fsync has a side effect of journal-

ing inode changes. Next, the reader attempts to refresh the inode by reading the relevant page of the inodes file, which requires obtaining the corresponding token. The second token request forces the writer to flush the inodes page and yield the token. Now the reader reads two disk blocks the inodes page and the file data block, thus completing the read access. In summary, it requires two token downgrades at the writer, involving disk writes to flush an inodes page, a journal page, and a data block. Our measurements indicate that these downgrades take a total of about 94 milliseconds, which is the major contributor to 118 milliseconds of elapsed time for the read access. Note that transfer of 4K byte page takes about 6 to 8 milliseconds, and about 12 milliseconds for disk I/O. The rest of time is in system overhead, CPU contention from the writer-initiated token activity, and disk seek time.

In PJFS, the write access requires one revoke on the reader node, which invalidates the cached inode and data. This revoke plus the rest of token upgrade takes about 24 milliseconds. Since the inode and data are still cached in the writer, there are no further disk I/O's involved. The write access elapsed time is about 58 milliseconds because of the additional CPU contention from token downgrade activity initiated by the reader.

In Calypso, the reader needs to downgrade only the file token on the writer. This forces the writer to flush the file contents to the server, where they are written to JFS cache. In the 8 bytes experiment, this takes about 11 milliseconds. On the reader, 4 additional milliseconds are used for acquiring the token and for reading the data. The actual elapsed time is about 21 milliseconds, because of the CPU contention from writer-initiated revoke activity.

Finally, in Calypso, the writer takes about 14 milliseconds to revoke the token on the reader; because of the additional CPU contention from the reader-initiated downgrade, the total elapsed time for the write is about 22 milliseconds. When 4K bytes are written and read, the times increase to 26 and 27 milliseconds respectively because of the additional time needed for moving the extra bytes. Overall, Calypso benefits from the fact that only one token is being revoked or downgraded, and data exchange is going through the JFS cache.

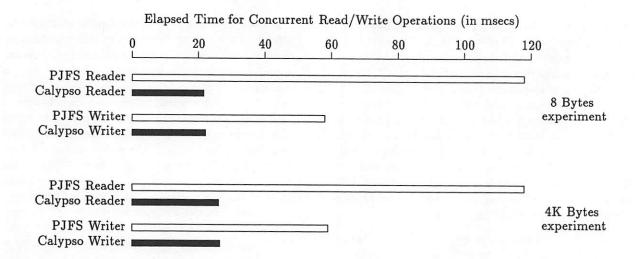


Figure 5: Results from the concurrent access measurements. Two sets of experiments are conducted, and in each set a reader and a writer simultaneously read and write the contents of the same file. The file contains 8 bytes in first experiment, and 4K bytes in the second.

6.5 Effects of Multi-Ported Disks

In the experiments above, we used the RLV mechanism to access the remote disk. However, since a multi-ported disk provides a direct connection to the disk, it can obviously improve access time. This is particularly true in the case of the micro benchmark and 2 Mbyte file read measurements discussed in Sections 6.1 and 6.3. As the hardware fan out is limited to four ports at this time, scalability measurements are not relevant.

However, a multi-ported disk can actually result in poor performance for the concurrent benchmark unless the disk uses non-volatile RAM cache. Without the NVRAM cache, the reader in the concurrent test experiences a latency of 24 milliseconds to read a 4K byte block (12 milliseconds for the writer to flush it to the disk, and another 12 milliseconds for the reader to read it). When using the RLV, the latency is only 16 milliseconds because of the buffering in the RLV server. Not all multi-ported disks use NVRAM cache. Given the fan out limitations and poor write-sharing performance, a general use of multi-ported disks may be in tolerating node failures.

6.6 Generalization Beyond JFS

How dependent are these results on the fact that PJFS uses JFS as the basis? Clearly, the dependency is significant. However, performance similar to PJFS can be expected from any physical file that treats metadata (e.g., inodes) separate from actual data, and most UNIX file systems do so. JFS's journaling also contributes to PJFS's poor performance in the concurrent test. But, without the journaling, JFS (and hence PJFS) performs poorly for typical interactive workloads because of the need to write small amounts of data to the disk. The important insight from these measurements is that file sharing as opposed to data sharing is the most efficient method. The shared file system approach provides caching and hiding of the internal representation to help this.

7 Conclusion

This paper evaluated two design alternatives to a file system targeted for clusters such as the IBM Scalable POWERparallel and HACMP. Since a cluster consists of homogeneous processors, two possible design alternatives were considered: a shared disk approach where serialized, multiple instances of a single-system file system directly access file data as disk blocks; and a shared file system approach, which is the conventional method of distributing file system function between a client and a server using the vnode interface. While most network file systems such as NFS and AFS follow the shared file system approach, the VAXCluster file system employs the shared disk approach. The work reported here is the first to compare the two approaches. For an even comparison, both approaches have been implemented using the same cache consistency mechanism and cache manager. The implementation experience and performance

measurements are the basis for the evaluation reported here.

Our implementation experience suggests that the serialization of metadata accesses is the most difficult in the shared disk approach, unless the single-system file system is designed with cluster usage in mind. As we extended AIX single-system file system to the cluster, single-system optimized metadata design and journaling of metadata changes turned out to be most difficult to handle.

Performance measurements of caching efficiency, scalability, throughput, and write-sharing show that the shared file system approach has two advantages: about 2 to 4 times better performance in write-sharing, and about 30% better read throughput. The shared disk approach, however, has a small advantage when almost all accesses can be serviced by the local file cache. Measurements also indicate that, in the shared-disk approach, multi-ported disks do not improve write sharing performance unless the disks use NVRAM caching. The overall conclusion is that the shared file system approach is the right direction for a cluster file system. It should exploit cluster hardware for performance and high availability, and provide efficient support for transactional as well as interactive workloads.

Acknowledgments

We thank the anonymous referees, in particular, our shepherd Peter Honeyman, for helping to improve the paper significantly. We gratefully acknowledge the contributions of the rest of the Calypso team: Bill Kish, Ananda Rao Ladi, Andy Zlotek, Edie Gunter, and Arup Mukherjee. Comments and input from Bob Curran, Wayne Zakaras, Donavon Johnson, and Damon Permezel during the Calypso project have been invaluable.

References

- [1] IBM Corporation. 9076 Scalable POWERparallel 2: Technical Brochure; Order Number: GH23-2485-00, May 1994. Information available from http://ibm.tc.cornell.edu Web page.
- [2] IBM Corporation. AIX HACMP/6000 System Overview Ver. 2.1.0; Order Number: SC23-2595-00, December 1993. Available from IBM Branch Offices.

- [3] N. P. Kronenberg, H. M. Levy, and W. D. Strecker. VAXClusters: A Closely-Coupled Distributed System. ACM Transactions on Computer Systems, 4(2), May 1986.
- [4] POSIX (IEEE Std 1003.1-1990 ISO/IEC Std. 9945-1: 1990). Portable Operating System Interface (POSIX) Part 1: System Application Program Interface. IEEE, 1990.
- [5] A. Mohindra and M. Devarakonda. Distributed Token Management in Calypso File System. In Proc. of IEEE Symp. on Parallel and Distributed Processing, Oct 1994.
- [6] A. Chang, M. F. Mergen, R. K. Rader, J. A. Roberts, and S. L. Porter. Evolution of Storage Facilities in AIX Version 3 for RISC System/6000 Processors. IBM Journal Research and Development, 34(1), January 1990.
- [7] M. Devarakonda, B. Kish, and A. Mohindra. Non-Disruptive Recovery in Calypso File System. Technical Report RC 19794, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10591, Oct 1994.
- [8] W. E. Snaman, Jr. Application Design in a VAXCluster System. *Digital Technical Jour*nal, 3(3), Summer 1991.
- [9] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In Proc. of the 10th Symposium on Operating System Principles, 1985.
- [10] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a Distributed File System. In Proc. of the 13th Symposium on Operating System Principles, 1991.
- [11] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network File System. In Proc. of Summer USENIX Conference, 1985.
- [12] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed File System. ACM Transactions on Computer Systems, 6(1), February 1988.
- [13] M. L. Kazar and others. DEcorum File System Architectural Overview. In Proc. of Summer USENIX Conference, June 1990.

- [14] G. J. Popek and B. J. Walker. The LOCUS Distributed System Architecture. The MIT Press, 1985.
- [15] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. ACM Transactions on Computer Systems, 6(1), February 1988.
- [16] A. Goldstein. The Design and Implementation of a Distributed File System. *Digital Technical Journal*, (5), September 1987.
- [17] B. Welch. A Comparison of Three Distributed File System Architectures. Computing Systems, 7(2), Spring 1994.
- [18] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In Proc. of Summer USENIX Conference, 1986.
- [19] VERITAS Software, Santa Clara, CA 95054. VERITAS VxFS File System, Jan 1994.
- [20] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The Episode File System. In Proc. of Winter USENIX Conference, Jan 1992.
- [21] Maurice J. Bach. The Design of the UNIX Operating System. Prentice-Hall, 1986.

Murthy Devarakonda received his Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in January 1988. Since then he has been with the IBM Thomas J. Watson Research Center, Yorktown Heights. At the present, he manages the Object-Oriented Systems Technology group focusing on distributed data management, object-orientation in systems software, and object-support systems. His electronic mail address is mdev@watson.ibm.com.

Ajay Mohindra received his Ph.D. in Computer Science from Georgia Institute of Technology in 1993. Currently, he is a Research Staff Member in the Object-Oriented Systems Technology group at the IBM Thomas J. Watson Research Center, Yorktown Heights. His research interests include computer architecture, operating systems, and distributed systems. He can be reached at a jay@watson.ibm.com.

Jill Simoneaux was a member of the Calypso project at the IBM Thomas J. Watson Research Center, Yorktown Heights. She is now employed

by the Intel Corporation. Her electronic mail address is jill_simoneaux@ccm.hf.intel.com.

William H. Tetzlaff is currently Senior Manager of Operating System Technology at the IBM Thomas J. Watson Research Center, Yorktown Heights. He received his MS in Computer Science from Polytechnic University and his BS from Northwestern University. He is a Senior Technical Staff Member at IBM and a member of the IBM Academy of Technology. His electronic mail address is tetzlaf@watson.ibm.com.

MULTI-RESIDENT AFS: AN ADVENTURE IN MASS STORAGE

Jonathan S. Goldick, Kathy Benninger, Christopher Kirby, Christopher Maher, Bill Zumach Pittsburgh Supercomputing Center

1. Abstract

The Pittsburgh Supercomputing Center has been working to integrate distributed file system technology with hierarchical mass storage. We produced a system utilizing the Andrew File System that can be interfaced to many mass storage systems. We retained the semantics of AFS and compatibility with standard clients and servers. The architecture has a logical separation between the facility that provides the user interface and access semantics and the management of the storage systems that contain user data. Support for file level replication is provided for high availability to data in a fashion that is transparent to users. This system is called Multi-Resident AFS.

2. Introduction

There has been a great deal of work in the last ten years in making distributed file systems faster, more functional, and easier to use. This has lead to their widespread use from the desktop to the supercomputer. The explosive growth of data being stored in these systems, combined with the exponential growth in client capacity, emphasizes a necessary capability, support for hierarchical mass storage. The Pittsburgh Supercomputing Center has been working for several years to address this problem. Our experiences with our system based on the Andrew File System (AFS) from Transarc will be described. While the paper will concentrate on AFS, the lessons learned apply to most, if not all, distributed file systems.

3. The Problem

In 1990, the Pittsburgh Supercomputing Center was facing a serious problem in providing access to the large volume of data produced by our users. We had thousands of users spread out across the United States and needed a way to provide them with easy access to

the data they produce on the center's supercomputers and as a vehicle for collaboration with other scientists. While we were already running the Los Alamos Common File System (CFS) [1] to service our mass storage needs, it had only a simple get/put file interface and no distributed access.

We identified distributed file systems technology as a way to provide our users with both a friendly interface and the ability to access their data from their home sites. However, there were few products available that offered both distributed access and support for high latency, high capacity media like tape. The products we did find offered FTP and NFS interfaces but suffered from several significant problems. When tape mounts weren't completed within some minimum time, users would see NFS time-outs. All of the software had to be run on a single, high capacity server machine; there was no support for distributing the load across several file servers without duplication all of the equipment. This inability to spread the load over multiple servers was made more painful by the fact that the NFS interface offered by these products had to query the servers to check the consistency of any data they might have cached. This was a major performance problem considering that our clients were much faster than any server available. There was also very little in the way of security in these systems. They were designed with the model that every machine and network between the data and users was under a centralized control. Such an environment is becoming increasingly rare and certainly didn't include our site.

Some other, more long range, problems we observed in the available products included the lack of options for server platforms. These systems were generally designed for a specific machine and there was little choice in hardware. This was made even worse by the observation that there were also no standards to allow us to move data from one vendor's system to another, thereby locking us into a single product line.

While the amount of data we store is much larger than most sites, our problem of not being able to afford as much magnetic disk as our users want is universal. The need for secure, distributed access to data from many clients is not limited to supercomputing centers.

4. Existing Solutions

There are a number of ways to approach this problem. Some systems extend the UNIXTM file system at the kernel level to intercept file I/O and do the necessary work to bring the data onto a disk. Examples of this method include NAStore from NASA Ames [2], InfiniteStorage from Epoch Systems [3], and CRAY Data Migration Facility from CRAY Research. The file system is periodically scanned for unused files which are moved to slower, cheaper media. Distributed access to the data is generally provided by NFS exporting the local file system with the resulting potential for NFS time-outs mentioned earlier. For those systems not provided by the vendor of the operating system, there is the even bigger problem of maintaining the kernel modifications across versions of the operating system and across hardware platforms.

Another approach to solving this problem is to let the mass storage system export an NFS server interface without an explicit UNIX file system underneath it. The file data is spread out among disks and tapes but appears as a single NFS mounted partition to the users. The file name space is kept in some sort of database. UniTree [4], which is provided by several vendors, uses this approach. This has the advantage of a standardized, user-mode interface running on the server. There are none of the problems typically associated with changing a kernel to provide the necessary interface. The down side of this approach is that the mass storage vendor has to reproduce the UNIX file systems technology in user mode that vendors provide in the kernel. The caching strategies and use of kernel memory and buffers are not available to them. This causes a general performance problem when doing operations that don't involve much actual movement of data.

More recently, the Center for Information Technology Integration at the University of Michigan proposed the mass store itself be considered the file system, not just a bag on the side of a disk-based file system [5]. This is based on the approach taken by the Plan 9 [6] file system where high-speed RAM acts as a cache for magnetic disk, which in turn acts as a cache for WORM optical disk. Disks and memory would be used as high speed caches in front of the relatively slow tape or optical mass storage system. The idea is

to create a transparent byte-level interface to files located in the mass storage system. A major problem with this approach becomes apparent when file system meta-data is considered. File systems generally do many small, random access operations on meta-data files, such as directories. If a directory file is not in the disk or memory cache, performance is very poor when referencing any file below it in the name space. Given that tape systems are rarely random access and have generally poor start/stop behavior, meta-data operations couldn't be handled until the file had first been moved to the disk or memory cache. The implications of this are more clear when one considers what it would be like to fsck such a file system. That operation performs random access reads to every meta-data file in the system to check its integrity.

5. Why AFS?

We chose AFS [7][8] for our distributed file system for a number of reasons. From a user's perspective it has several desirable features including a global name space where the path to a file is the same from any client, Kerberos V4 security, access control lists on each directory, semantics reasonably close to UNIX file system semantics, and good performance using client-side caching. Some of the important management features include the volume abstraction. location transparency for volumes, read-only volumes, replication of volumes across multiple servers, and volume quotas. We also observed that AFS scaled well; its servers could support a much larger number of clients per machine than NFS. This is due to its use of callbacks to inform clients when data they had cached became invalid. This eliminates the need for clients to make RPCs to the servers whenever they want to validate some object they have cached.

6. Mass Storage Extensions to AFS

The most fundamental goal of our extensions is to preserve the semantics of AFS and not attempt to make it an expert in mass storage systems. By leveraging each type of technology that is available, we provide a framework to allow each component to do what it does best. The file system handles the user interface and access semantics, and the mass storage system handles the bytes. The logical separation is that the file system handles the meta-data, and the mass storage system handles the user data. The system we created to implement this model is named Multi-Resident AFS.

Many changes are required to make AFS function as part of a true mass storage system. The main design goals are hierarchical storage management, fault

tolerance, and access to existing technology, while still retaining compatibility with standard AFS clients and servers. We achieved the former goals through extensive generalizations of the AFS server code, and the latter by working within the standard AFS data structures.

7. Multiply-Resident Data

Our extensions allow AFS to move from a system capable only of serving UNIX disk partitions to a true hierarchical storage system. Where files were once restricted to exist in only one place, they are now able to exist in up to 32 separate locations. These new copies of data, known as residencies, give AFS many advantages. These additional residencies not only protect AFS against media failure, but also provide high availability of user data. If a storage system is unavailable for preventive maintenance or any other reason, AFS will automatically obtain the data from the highest priority storage system that is available. This is all completely transparent to users. In this manner, we can take preventive maintenance time with minimal impact on our users.

This functionality is similar, but orthogonal, to AFS volume replication. Here we are talking about a perfile replication, where each file is treated independently. In AFS volume replication, all of the files within a volume are copied at the same time and to the same destination.

The ability to have multiple copies of file data becomes more important as support for mass storage is added. A single 8 mm tape can hold 50,000 files with an average size of 100 KBytes. The loss of this single tape for any reason would have an enormous impact on users. As file servers gain the ability to utilize ever larger capacity media, this danger grows. This applies to disks as well as tape, given how fast the storage densities of disks are growing.

Figure 1 shows how an AFS volume, the management unit for an aggregate of files, changes with this additional support. Notice that files can be put on different storage systems based on their sizes.

8. Consistency

Allowing a file to exist in more than one storage system brings up the problem of insuring that each copy contains the same data. When a multi-resident file is modified, only the new, modified copy remains. All of the references to copies of the previous version of the file are removed. This new version of the file is then treated the same as any new file in terms of data migration. The file server also

guarantees that no partially created copy of a file will ever be referenced.

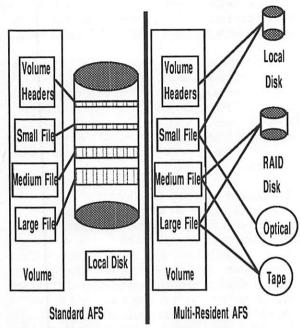


Figure 1. Volume layout

9. Data Migration

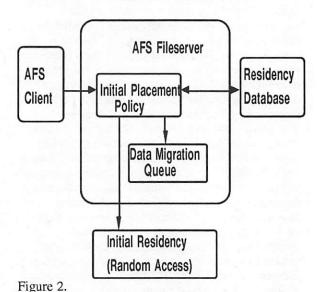
With the addition of multi-resident support, it is necessary to have a data migration system to add and remove these additional residencies. There is a small replicated residency database which contains information about the known storage systems, but not volume or file level information. For each storage system, the information includes a relative priority, a desired file size distribution, the list of machines that offered the storage system, and various data migration policy variables. The goal of the data migrator is to optimize the use of storage by transparently moving data between members of the storage hierarchy. This includes adding redundant copies of data to provide fault tolerance, removing copies of data from storage systems whose free space fell too low, moving hot files to faster media, and moving cold files to slower media. Another important task of the data migrator is to insure that each storage system has the desired file size distribution. This way we can tune each storage system to a specific target range of file sizes, making it much more efficient. The policies governing these actions can be changed easily, either by changing the contents of the residency database or by changing a configuration file. All policy variables in the system are dynamic and can be changed without interruption of service.

The data migrator also has the ability to move files or whole volumes to off-line storage. These file residencies are unavailable to users but can be migrated back by the site manager. This feature is intended to handle the case where a collection of data is no longer in use, but may be needed at a later date.

It should be noted that while this system moves data between storage systems, it has no knowledge of what those storage systems do with the data. For example, AFS might create a residency in an NFS archival system in which the data initially lands on a disk and is later moved to an optical platter. AFS has no need to know that the file has been moved within the NFS storage system.

Another important responsibility of the data migration system is to create random access residencies of files on demand. AFS clients access files from the AFS server in a random access fashion. If the file does not exist on a random access storage system when a fetch request is received, the data migration system must create an additional residency. We call this process "spooling a file", and we call storage systems that do not offer random access "archival" storage systems. This ability to spool a file allows AFS to make use of storage systems for which random access is either not available, or too slow to be practical.

File Creation



10. Data Flow

At this point we will describe how data is moved through the storage systems. Figure 2 shows how a file is created. A storage system is chosen based on the file's size and the free space on the random access storage systems. The residency database is consulted to determine which storage system is the best choice for this new file. This newly created file is also added to a data migration queue for possible creation of additional residencies on archival storage.

Figure 3 shows how this data migration queue is processed into requests to copy the file to one or more archival storage systems. The residency database has a policy variable for each archival storage system that indicates how old a file must be before it gets an additional copy made there. Setting the minimum age to a small value reduces the window of vulnerability to media failures, but this also increases the likelihood that a temporary file will be moved to archival storage unnecessarily. For our environment we have found a minimum age of six hours to be a reasonable value.

Data Migration to archival storage

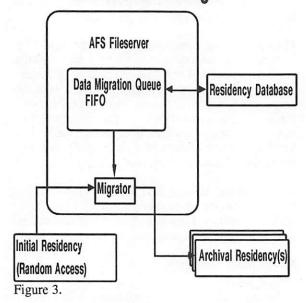


Figure 4 shows how space is made available when there is a shortage. There is a new process that runs on each file server machine which checks storage systems for insufficient free space every five minutes. It processes the volume meta-data and bins up files that are candidates for removal from each storage system. A primary weighting function is used to determine into which bin a file should be placed and a

secondary weighting function is used to sort the files within a bin. These weighting functions are configurable, we use seconds since the last user access as the primary weight and file length in bytes as the secondary. The deletion policy also includes the ability to provide a list of free space thresholds to be applied to different primary weight ranges.

Once the bins of candidate files are created, the free space reported by the residency database is compared to the threshold that should be applied to the highest weight bin. If there is insufficient free space on a particular storage system, requests to remove candidate files from that storage system are sent to the file server until we reach a stopping threshold. These requests must go through the file server to preserve the AFS locking semantics.

One of the major advantages of using last access time as the primary weight is that it eliminates the need to process the volume meta-data every time the checker wakes up. Any bins of candidate files that remain from the last run are still valid. The highest weight bin from the last run is still the highest weight bin because all files age at the same rate. This would not be true if the primary weight was file length.

Space Reclaimation

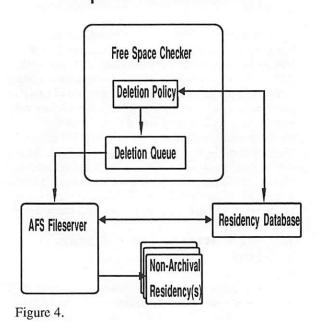
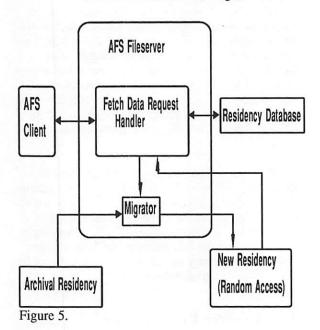


Figure 5 shows how a file is spooled from a nonrandom access storage system to a random access storage system when a client tries to read it. As mentioned previously, the FetchData RPC requires a random access copy of the file. When the RPC is processed, the file server looks for a random access

copy of the file. When one isn't found, a new copy is made on a storage system chosen by the residency database, the decision being based on file size and storage system priority. Once the new copy has been created, the FetchData RPC continues. The new copy is not considered valid, and is not referenced, until it has been made completely. This is intended to prevent a file from having multiple residencies with different data.

Data Migration from archival storage



One of the other interesting features of this data migration architecture is that by adding additional file residencies as soon as a file has reached a minimum age, we can free up space on our random access storage systems at the speed of a delete operation. Many systems don't begin to create additional residencies until the random access storage system starts to get full. That can often lead to a situation where free space can only be made available at the speed at which one can write data to the generally slower archival storage system. Also, unlike most other architectures, when one modifies a file there is no requirement that it be on random access storage before the modification can begin. Most products spool a file to a disk first, which is wasted effort if the modification involves a complete overwrite of the file. Our system treats the new version of a file as a separate copy of data and will only access the previous version in the event that some part of the old file was not overwritten. This also avoids getting into the situation where a file whose old version was

small and whose new version is large ends up residing on a storage system reserved for small files.

Considering the exponential growth of data being stored, it becomes very difficult, as well as expensive, to do backups. The ability to create additional residencies for each file helps a great deal with this backup problem by offering a new level of fault tolerance. Our extended AFS system essentially provides a continuous, per-file, backup system. This has enormous performance advantages over the usual snapshot backup systems which operate on an aggregate of files. Since every file is being treated independently, there is no need for the system to be quiescent to insure self-consistent meta-data. Additionally, we can control which files are backedup, and to where, based on characteristics such as size and access history. However, this does not handle the case where a user deletes a file by accident. The backup volume ability in AFS provides some limited recovery from this failure, but backup volumes are often remade each night. If a user does not realize the mistake in time, the backup volume will not help. More will be said on this point in the lessons learned section.

Another important advantage of multiply-resident files is that we are able to bring new storage systems into service and take old ones out of service in a user transparent fashion. It is often the case that when replacing a major storage system, there is no simple way of moving the data from the old system to the new one. With only two directives, our AFS system will do the migration automatically.

11. Generic I/O Interface

Many of the storage systems we wanted to integrate with AFS did not offer the UNIX file system interface required by the AFS open-by-inode I/O calls. To solve this, we generalized the I/O system of AFS. By doing all I/O through a generic interface, the UNIX file system dependence of AFS is removed. This allows AFS to take advantage of the available technology without having to reinvent it. In order to interface to a given storage system, only a small I/O library is required. AFS can then access a wide variety of storage systems, including UniTree, CRAY Data Migration Facility, Epoch, and a Maximum Strategy RAID disk system.

There are three types of routines that must be provided when creating an I/O interface. The first group of routines does file management. Open, Read, Write, Seek, Truncate, CloseIfOpen, Stat, and FsyncFile are generally well understood routines so we won't describe them here, except to say that an

interface that doesn't support random access would not need to actually implement a seek routine. The other file management routines include IncDec and BulkIncDec, which are used to change the volume reference count on file data, and GenerateLookupTags, which is used to create a 64-bit identifier for a new file on a storage system. The first two routines are similar to link and unlink, and the last routine is similar to create.

The next group of routines are storage system management routines. They include GetFileSystems, GetFsAdvisoryLock, ReleaseFsAdvisoryLock, StatFs, VerifyResidency, ListFiles, ListVolumeHeaders, and Configure. These routines can be thought of as operating at the UNIX partition level. GetFileSystems is similar to returning a mount table. GetFsAdvisoryLock and ReleaseFsAdvisoryLock are used to control access to a storage system during execution of a salvage operation, the AFS equivalent of fsck. StatFs returns the free and total space available on a storage system. VerifyResidency reports whether a storage system is available for use. ListFiles is used to dump the list of files that exist on a storage system, this is equivalent to dumping the inode table on a UNIX file system. ListVolumeHeaders returns the list of AFS volumes that exist on a given local disk storage system. Finally, Configure is used to set up a storage system when it is first brought into service.

The last type of routine is the optional Import function. It is necessary to address the problem of how to merge an existing mass storage system with AFS. The Import routine allows one to create a file in the AFS name space and map it to a pre-existing piece of user data. It is intended that this routine will just alter the lookup handle of the user data to conform with the AFS naming scheme. No actual movement of the user data should be involved. This can be thought of as a rename operation, but not a copy. This feature also allows one to store data into a storage system shared with AFS, using some other mechanism, and later add it into the AFS name space.

12. AFS Servers Share Storage Systems

Up to this point we have only discussed a single AFS server environment. But in practice we have several servers, each with its own set of storage systems. Because of the expense involved in providing every server with direct access to every storage system, we provide a mechanism that allows the servers to share their storage systems. This mechanism, known as the remote I/O server, allows one AFS server to execute I/O calls on a storage system that is not directly connected to it. The remote I/O server is a

small RPC service that allows a machine to offer to AFS servers all of the storage systems to which it has direct access. This has the added advantage of allowing us to bring more machines into the server hierarchy without having to run full AFS file servers on them. As long as a machine is capable of running an AFS RPC service, it can offer storage systems to AFS file servers. It should be noted that each file server has a set of storage systems defined to be its "local disk" which only it may use. This "local disk" system corresponds directly with the standard AFS UNIX disk partitions in that data that is private to a server is kept here.

13. Salvaging

Salvaging consists of comparing the meta-data files, including directories, with what actually exists out on the various storage systems. Our system has a rewritten salvager to deal with the new problems that arise in this system. Now that files and directories can be spread out across multiple storage systems and multiple servers, the salvage process becomes much more complex. It is quite possible that when a salvage is being executed, not every machine or storage system is available. The new salvager is smart enough to handle this case without removing references to files whose existence can't be verified.

Another major issue is salvaging directories that are not on a random access storage system. The salvager must read the contents of every directory so that files can be claimed. Any files that have not been claimed need to be added to a lost+found directory. Many hierarchical storage products do not allow directories to be migrated off of the local disk because of the need to have them available for salvaging in a random access fashion. The new salvager gets around that limitation by guaranteeing that directory check operations do no seeks and therefore do not require random access. This allows directories to be migrated off of the disks without the need to spool them back from tape whenever the salvager runs.

There is also support in the salvager for doing selective salvages of storage systems. As a rule, we trust our archival storage system not to lose files. Therefore we don't include it in storage systems we salvage by default. This saves a great deal of time.

14. Comparison With Standard AFS

In light of what has been described above, we will now take a look at how some fundamental AFS concepts have changed. In standard AFS, the logical unit of files, a volume, is tied to a specific UNIX

partition on a specific server. This has been extended to allow the volume to span many storage systems, as shown in Figure 1. By limiting only the volume headers to the original server and partition, we can effectively utilize a hierarchy of storage systems for user data, while retaining backwards compatibility with standard AFS volume utilities. Previously, one had to worry about assigning too many volumes to a single UNIX partition and the degree to which its space was being over-allocated. In the new model, this is no longer a consideration.

Standard AFS has the concept of a read-only volume that is a replica of a read-write volume that may exist on another server and/or partition. These read-only volumes are used to distribute the load to a very active, and slowly changing, collection of files among a number of AFS servers. They also provide a limited amount of fault tolerance. If one replica is unavailable, AFS clients will automatically fail over to another replica. The improvement we have made to this system is that the read-only replica(s) and the read-write volume actually share copies of data in the shared storage systems, even if the volumes reside on different servers. In this manner we gain all of the advantages of having replicated volumes without paying the storage penalties. It is important to remember that we no longer use replicated volumes to protect against media failure; the multiple file residencies take care of that. Therefore we lose nothing by having the replicas share copies of data. The replicated volumes are used to spread out the load of AFS name space operations.

The lookup information for files in standard AFS is kept in one of two volume meta-data files in a data structure called a vice node (vnode). In order to support multiple residencies of a file, while working within existing data structures, we create chains of vnodes. The original vnode of a file is unchanged except that two previously spare fields are used to point to other vnodes. These other vnodes hold the lookup information for additional file residencies. This approach allows system utilities and dump formats to be backwards compatible.

Standard AFS keeps track of very little in the way of a file's usage history. We add a per-file access history with the addition of another vnode to the chain. This history includes details of the most recent accesses and keeps various counters that let us know how well the system is managing each file. This information is used by the data migration system to determine which files should be migrated and to where. We also provide reporting tools that generate reports based on this information.

15. Current System Usage

Our current mass storage system is shown in figure 6. We use the local disk and the Maximum Strategy RAID disk banks for random access storage and CRAY Data Migration Facility (DMF) for archival storage. We configured our residency database to put files 64 KBytes or less on the local disk residency and larger files on the RAID disks. Files that are over 4 KBytes will be migrated to DMF.

The AFS clients range from hundreds of workstations to a CRAY C90. The bulk of the files are created by the workstations and the bulk of the data is created by the CRAY. The workstations include many personal workstations and a few dozen members of workstation farms doing large scale data processing.

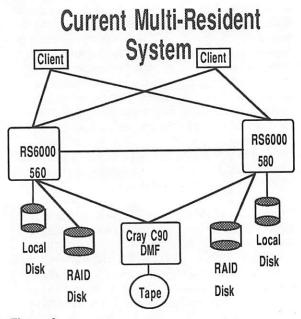


Figure 6.

Figures 7 though 10 describe how this system is being used. Figure 7 shows that our average file size is somewhat larger than the typical UNIX file [5][9].

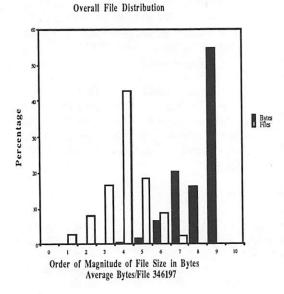


Figure 7.

Figures 8 and 9 show how the system sends small files to the local disk and larger files to the RAID disk banks.

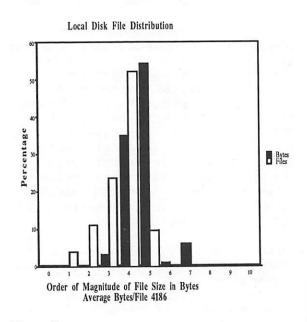


Figure 8.

RAID Disk File Distribution

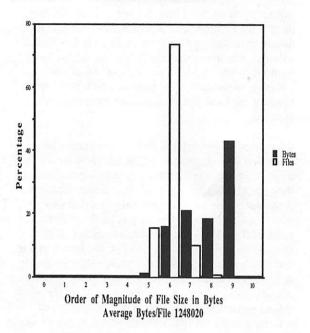


Figure 9.



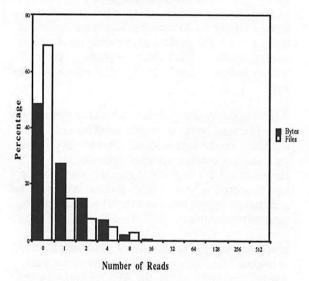


Figure 10.

Figure 10 shows that almost seventy percent of the files, about half the data, are written but never read back from the file server. This should not be taken to mean that the file server processes mostly write requests. It demonstrates that a large amount of our files and data are not used for longer than they can remain in our AFS clients' caches. This can be

attributed to both the AFS client caching mechanism and the fact that our clients have relatively large cache sizes.

As of 7/5/94 the system managed 43434 directories, with 100 MBytes, 591044 files, with 221 GBytes, and 18736 symbolic links, with 338 KBytes.

16. Lessons Learned

File servers with integrated mass storage support can service much more data per server than conventional servers. These systems not only save money by reducing the amount of disks needed, but can reduce the number of file servers needed as well. The number of disks that can be attached to each file server becomes less of a consideration. The issue of how many file servers are needed to meet the transaction load remains. Currently, we service as much data from two RS/6000 servers running this software as we can service with six DecStation 5000 servers running standard AFS.

As the space available for storing files increases, so does the average file size. Between November 1992 and July 1994 the average file size has grown from 51 KBytes to 338 KBytes. However, the median file size has not changed. This tells us that users are making use of the fact that they can now store more large files than ever before, but the typical file size hasn't changed.

File servers have more information on when a file is likely to be used again than the mass storage systems they use to store the bytes. Often the storage systems come with software which attempts to determine when files should be moved between its high and low performance media. The assumptions they use in that determination won't take into account the fact that the file server is caching the file on other storage systems that it may consider higher priority. and that AFS clients are also caching that data. For tape management systems, we find it best to issue a directive to indicate that it is relatively unlikely that it will receive any read requests for a file recently stored there. Such files are already on a random access storage system and probably in an AFS client's cache as well.

Many systems block writers when there isn't enough free space to continue. The idea is that files will be migrated to free up space and then the writers will be woken up. This does not work for AFS because there is a limited number of execution threads on the server and it ends up blocking out all client operations while it waits for space to be freed up. In addition, the AFS clients get stuck waiting for RPCs to complete to the server and AFS clients do not have the ability to

interrupt Fetch and Store operations. In this architecture, if the server could keep up with the demand, it wouldn't need to block. If it can't, there is no reason to expect that blocking the writer will accomplish anything in a reasonable amount of time. Allowing servers to share storage systems does a great deal in handling bursts in load, but when space runs out, a failure should be returned to the clients. A possible improvement to this would be the ability for clients to retry store operations later. This shouldn't be difficult as the client is already caching the file, but would require modifications to AFS client software.

As with all mass storage systems where unused files are not likely to be on low latency storage, users learn not to do certain operations. A user will rarely make the mistake of typing 'file *' across a bunch of old files more than once. This learning process no doubt reduces load to mass storage system and further reduces the likelihood that old files are ever read again.

Even though the file server has longer code paths than standard AFS, the performance for most operations is at least as fast. This is true because in our experience AFS performance is limited by the RX transport layer and the ability of the server to process its UDP packets. This has remained true in the latest release of AFS, version 3.3a, when using an FDDI network. The server CPU saturates when trying to send a large number of UDP packets per second over FDDI. Therefore the extra server code does not have a noticeable effect. It is also possible to get better performance than standard AFS because of the ability to use faster media and I/O interfaces that are optimized for a certain file size range. For example, our RAID-3 disks are much faster than our SCSI disks when transfering files over two megabytes in length. These RAID-3 disks can not be accessed from standard AFS because they do not have a UNIX file system structure. When running performance tests with a CRAY C90 client, an RS/6000-580 server, and an FDDI network, there is no measurable performance difference between standard AFS and Multi-Resident AFS when reading and writing files. Writes happen at about 1.2 MBytes/Sec. and reads at about 1 MByte/Sec.

There are two operations that are noticeably slower than in standard AFS, one being file deletion and the other salvaging. Since files generally reside in more than one storage system, it takes longer to delete them. We will be addressing this problem by making file deletion asynchronous. The user will get an immediate return once the RPC has been processed, but the actual file deletion will occur at a later time. This should make file deletion faster than in standard

AFS, at least from the users' perspective. The salvager is slower for several reasons. There are more storage systems for which the list of files residing there must be obtained. There is more volume metadata than in standard AFS which must be checked for validity. For the system described earlier, we have found it to take 1.5 hours to salvage all of the volumes in the cell. The bulk of this time is spent running the ListFiles routine across the storage systems.

It has been suggested that performance could be improved if the server pre-fetched a whole directory at a time. The idea is that if a user accesses a directory, they are highly likely to read files in that directory. Analysis of our file accesses indicates that while it's true that a file in that directory is likely to be read, most of the files won't be. The net effect of this strategy is to do much more I/O on the server and to use much more high priority storage than is strictly necessary. The overall performance of the system actually goes down. A mechanism that might allow this to work effectively would be the addition of affinity groups. A file could be added to one or more groups of related files to indicate that they are referenced as a group. This ability would have another potential performance improvement when the files are out on tape. Many mass storage systems that offer media with a high latency to mount make attempts to reorder requests so that they are grouped by tape, thereby reducing the number of mounts. Since AFS clients do not send bulk requests for file fetches to the server, there is currently no ability to take advantage of this feature. With the addition of affinity groups, related files could be put on the same tape.

When configuring the system, we had to choose how old a file must be before it gets an additional residency on our tape system. The shorter this time, the faster one can free up space when needed. However, if a file is going to be temporary, adding a tape residency is a waste of resources. We found that a minimum age of six hours seems to avoid having most of the temporary files added to archival storage.

It was necessary to do something to address the need to backup files in the event that a user accidentally deleted their data. Using the AFS backup volume to provide access to the previous night's file system handled most, but not all, cases. We did not have the option of doing full backups because of the scaling problem. Additionally, most of our data was out in archival storage, requiring a tape mount for almost every file just to dump its contents. There was no possibility of finishing volume dumps within 24 hours. What we chose to do was to dump the metadata for all files, but only the data that was resident

on the local disk. Since most of our files are small, they tended to land on the local disk and were included in the dump. For the larger files, we took advantage of the fact that our archival storage system supported both soft and hard deletion. When AFS files were deleted, they were still on a tape until an administrators did a hard deletion. Having the metadata for all files in the dumps allowed us to find files that had been deleted by users.

Currently, the data must still flow through the file servers. We are adding the ability to do third party transfers to eliminate this potential bottleneck. This will allow data to flow directly between the client and the storage system without passing through the file server's memory. This feature requires AFS client modifications so it will not be available to standard AFS clients.

17. Summary

The Multi-Resident AFS architecture successfully integrates a wide variety of mass storage systems with a distributed file system. Its ability to separate the user interface and access semantics from the mechanisms by which data is stored make the architecture very flexible. We believe it addresses the issues involved in bringing mass storage systems into more main stream environments.

18. Acknowledgments

UNIX is a trademark of Unix Systems Laboratories.

19. References

- [11 Collins, M. W., Mexal, C. W., "The Los Alamos Common File System," Tutorial Notes, Ninth IEEE Symposium on Mass Storage Systems, October 1988.
- [2] Tweten, D., "Hiding Mass Storage Under Unix: NASA's MSS-II Architecture," Tenth IEEE Symposium on Mass Storage Systems, pp 140-145, May 1990.
- [3] Foster, A., Habermehl, D., "Renaissance: Managing the Network Computer and its Storage Requirements," Eleventh IEEE Symposium on Mass Storage Systems, pp. 3-10, October 1991.
- [4] McClain, F., "DataTree and UniTree: Software for File and Storage Management," Tenth IEEE Symposium on Mass Storage Systems, pp. 126-128, May 1990.

- [5] Antonelli, C. J., Honeyman, P., "Integrating Mass Storage and File Systems," Twelfth IEEE Symposium on Mass Storage Systems, pp. 133-138, April 1993.
- [6] Quinlan, S., "A Cached WORM File System," Software Practice and Experience, Vol 21, No. 12, pp. 1289-1299, December 1991.
- [7] Morris, J.H., Satyanarayanan, M., Conner, M.H., Howard, J.H., Rosenthal, D.S.H., Smith, F.D., "Andrew: A Distributed Personal Computing Environment," Communications of the ACM, Vol 29, No. 3, pp. 184-201, March 1986.
- [8] Satyanarayanan, M., "Scalable, Secure, and Highly Available Distributed File Access", IEEE Trans. Computers, pp. 9-21, May, 1990.
- [9] Ousterhout, J., DaCosta, H.L., Harrison, D., Kunze, J., Kupfer, M., Thompson, J, "A Trace Driven Analysis of the Unix 4.2 BSD File System," Proceedings of the Tenth ACM Symposium on Operating Systems Principles, Orcas Island, December 1985.

20. Suggested Reading

"Mass Storage System Reference Model: Version 4," edited by Sam Coleman and Steve Miller, IEEE Technical Committee on Mass Storage Systems and Technology, May 1990.

"A Joint European Mass Storage Specification Effort," edited by Dick Dixon, European Centre for Medium-Range Weather Forecasts, Thirteenth IEEE Symposium on Mass Storage Systems, pp. 110-112, June 1994.

21. Biographies

Jonathan S. Goldick received a B.S. degree in Physics and a M.S. degree in Electrical Engineering from Carnegie Mellon University, Pittsburgh, PA, USA, in 1988 and 1989, respectively. He is currently a Senior Technical Specialist at the Pittsburgh Supercomputing Center, Pittsburgh, PA, USA. His research interests include the design, development, and analysis of distributed mass storage systems.

Kathy Benninger has been a hardware systems engineer with the Scientific Support Group at the PSC for four years. Her focus is on mass storage

system specification and integration and on supporting systems for scientific visualization. She received a BSEE from Carnegie Mellon University in 1984.

Christopher Kirby is a Senior Research Systems Programmer at the Pittsburgh Supercomputing Center and has worked there since 1990. He received a B.S. in Applied Math/Computer Science at Carnegie Mellon University in 1988 and a M.S. Computer Science from New York University in 1990. He was previously employed at At&T Bell Laboratories. His professional interests include design and development of mass storage and distributed file systems.

Christopher J. Maher is Director of Scientific Support at the Pittsburgh Supercomputing Center. He has been with PSC since 1988, as a Scientific Specialist, Scientific Support Manager and Scientific Support Director. At PSC he has over seen the installation of the Centers first mass storage system, supervised the development of multi-resident AFS, and is responsible for most of PSC's software development projects. Maher held postdoctoral fellowships at the Massachusetts Institute of Technology and Carnegie Mellon University prior to joining PSC's staff. He received is B.S., M.S. and Ph.D. from Carnegie Mellon University in Physics in 1980, 1982 and 1986 respectively. Maher has authored and co-authored numerous scientific publications.

Bill Zumach is a Senior Research Systems Programmer at the Pittsburgh Supercomputing Center, where he has been employed since 1992. He works on design and development of distributed files systems. He received a B.A in Mathematics from the University of Minnesota in 1987. He previously worked at the Astronomy Department of the University of Minnesota designing data collection and analysis software. His research interests include mass storage and distributed file system design and analysis.

RAMA: Easy Access to a High-Bandwidth Massively Parallel File System

Ethan L. Miller University of Maryland Baltimore County Randy H. Katz University of California at Berkeley

Abstract

Massively parallel file systems must provide high bandwidth file access to programs running on their machines. Most accomplish this goal by striping files across arrays of disks attached to a few specialized I/O nodes in the massively parallel processor (MPP). This arrangement requires programmers to give the file system many hints on how their data is to be laid out on disk if they want to achieve good performance. Additionally, the custom interface makes massively parallel file systems hard for programmers to use and difficult to seamlessly integrate into an environment with workstations and tertiary storage.

The RAMA file system addresses these problems by providing a massively parallel file system that does not need user hints to provide good performance. RAMA takes advantage of the recent decrease in physical disk size by assuming that each processor in an MPP has one or more disks attached to it. Hashing is then used to pseudo-randomly distribute data to all of these disks, insuring high bandwidth regardless of access pattern. Since MPP programs often have many nodes accessing a single file in parallel, the file system must allow access to different parts of the file without relying on a particular node. In RAMA, a file request involves only two nodes — the node making the request and the node on whose disk the data is stored. Thus, RAMA scales well to hundreds of processors. Since RAMA needs no layout hints from applications, it fits well into systems where users cannot (or will not) provide such hints. Fortunately, this flexibility does not cause a large loss of performance. RAMA's simulated performance is within 10-15% of the optimum performance of a similarly-sized striped file system, and is a factor of 4 or more better than a striped file system with poorly laid out data.

1. Introduction

Massively parallel computers are becoming a common sight in scientific computing centers because they provide scientists with very high speed at reasonable cost. However, programming these computers is often a daunting task, as each one comes with its own special programming interface to allow a programmer to squeeze every bit of performance out of the machine. This applies to file access as well; massively parallel file systems require hints from the application to provide high-bandwidth file service. Each machine's file system is different though, making programs difficult to port from one machine to another. In addition, many scientists use workstations to aid in their data analysis. They would like to easily access files on a massively parallel processor (MPP) without explicitly copying them to and from the machine. Often, these scientists must use tertiary storage to permanently save the large data sets they work with [7,15], and current parallel file systems do not interface well with mass storage systems.

Traditional MPPs use disk arrays attached to dedicated I/O nodes to provide file service. This approach is moderately scalable, though the single processor controlling many disks is a bottleneck. Recent advances in disk technology have resulted in smaller disks which may be spread around an MPP rather than concentrated on a few nodes.

RAMA addresses both ease of use and bottlenecks in massively parallel file systems using an MPP with one or more disks attached to every node. The file system is easily scalable: a file read or write request involves only the requesting node and the node with the desired data. By distributing data pseudo-randomly, RAMA insures that applications receive high bandwidth regardless of the pattern with which the data is accessed.

2. Background

Massively parallel processors (MPPs) have long had file systems, as most applications running on them require a stable store for input data and results. Disk is also used to run out-of-core algorithms too large to fit in the MPP's memory. It is the last use that generally places the highest demand on file systems used for scientific computation [14].

2.1. Parallel File Systems

Early MPP file systems made a concerted effort to permit the programmer to place data on disk near the processor that would use it. This was primarily done because the interconnection network between nodes was too slow to support full disk bandwidth for all of the disks. In the Intel iPSC/2 [16], for example, low network bandwidth restricted disk bandwidth. The Bridge file system [4], on the other hand, solved the problem by moving the computation to the data rather than shipping the data across a slow network.

Newer file systems, such as Vesta [2] run on machines that have sufficient interconnection network bandwidth to support longer paths between disked nodes and nodes making requests. These file systems must still struggle with placement information, however. Vesta uses a complex file access model in which the user establishes various views of a file. The file system uses this information to compute the best layout for data on the available disks. While this system performs well, it requires the user to tell the system how the data will be accessed. Vesta provides a default data layout, but using this arrangement results in performance penalties if the file is read or written with certain access patterns. Supplying hints may be acceptable for MPP users accustomed to complicated interfaces, but it is difficult for traditional workstation users who want their programs to be portable to different MPPs.

The CM-5 sfs [11] is another example of a modern MPP file system. The CM-5 uses dedicated disk nodes, each with a RAID-3 [8] attached, to store the data used in the CM-5. The data on these disks is available both to the CM-5 and, via NFS, to the outside world. The system achieves high bandwidth on a single file request by simultaneously using all of the disks to transfer data. However, this arrangement does not allow high concurrency access to files. Since a single file block is spread over multiple disks, the file system cannot read or write many different blocks at the same time. This restricts its ability to satisfy many simultaneous small file requests such as those required by compilations.

A common method of coping with the difficulties in efficiently using parallel file systems is to provide additional primitives to control data placement and manage file reads and writes efficiently. Systems such as PASSION [1] and disk-directed I/O [9] use software libraries to ease the interface between applications and a massively parallel file system. These systems rely on the compiler to orchestrate the movement of data between disk and processors in a parallel application. While this solution may work well for specialized MPP applications, though, it does not allow the integration of parallel file systems with the networks of workstations used by scientific researchers. The compute-intensive applications that run on the parallel processor may get good performance from the file system, but files must be explicitly copied to a standard file system before they can be examined by workstation-based tools.

Another shortcoming of parallel file systems is their inability to interface easily with tertiary storage systems. Traditional scientific supercomputer centers require terabytes of mass storage to hold all of the data that researchers generate and consume [6]. Manually transferring these files between a mass storage system and the parallel file system has two drawbacks. First, it requires users to assign two names to each file — one in the parallel file system, and a different one in the mass storage system. Second, it makes automatic file migration difficult, thus increasing the bandwidth the mass storage system must provide [15].

2.2. Parallel Applications

Parallel file systems are primarily used by computeintensive applications that require the gigaflops available only on parallel processors. Many of these applications do not place a continuous high demand on the parallel file system because their data sets fit in memory. Even for these programs, however, the file system can be a bottleneck in loading the initial data, writing out the final result, or storing intermediate results.

Applications such as computational fluid dynamics (CFD) and climate modeling often fit this model of computation. Current climate models, for example, require only hundreds of megabytes of memory to store the entire model. The model computes the change in climate over each half day, storing the results for later examination. While there is no demand for I/O during the simulation of the climate for each half day, the entire model must be quickly stored to disk after each time period. The resulting large I/Os, are large and sequential.

Some applications, however, have data sets that are larger than the memory available on the parallel processor. These algorithms are described as running outof-core, since they must use the disk to store their data, staging it in and out of memory as necessary. The decomposition of a $150,000 \times 150,000$ matrix requires 180 GB of storage just for the matrix itself; few parallel processors have sufficient memory to hold the entire matrix. Out-of-core applications are written to do as little I/O as possible to solve the problem; nonetheless, decomposing such a large matrix may require sustained bandwidth of hundreds of megabytes per second to the file system. Traditionally, the authors of these programs must map their data to specific MPP file system disks to guarantee good performance. However, doing so limits the application's portability.

3. RAMA Design

We propose a new parallel file system design that takes advantage of recent advances in disk and network technology by placing a small number of disks at every processor node in a parallel computer, and pseudo-randomly distributing data among those disks. This is a change from current parallel file systems that attach many disks to each of a few specialized I/O nodes. Instead of statically allocating nodes to either the file system or computation, RAMA (Rapid Access to Massive Archive) allows an MPP to use all of the nodes for both computation and file service.

The location of each block in RAMA is determined by a hash function, allowing any CPU in the file system to locate any block of any file without consulting a central node. This approach yields two major advantages: good performance across a wide variety of workloads without data placement hints, and scalability from fewer than ten to hundreds of node-disk pairs. This paper provides a brief overview of RAMA; a more complete description may be found in [13].

RAMA, like most file systems, is I/O-bound, as disk speeds are increasing less rapidly than network and CPU speeds. While physically small disks are not necessary for RAMA, they reduce hardware cost and complexity by allowing disks to be mounted directly on processor boards rather than connected using relatively long cables. High network bandwidth allows RAMA to overcome the slight latency disadvantage of not placing data "near" the node that will use it; thus, RAMA requires interconnection network link bandwidth to be an order of magnitude higher than disk bandwidth; this is currently the case, and the gap

in speeds will continue to widen. Network latency is less important for RAMA, however, since each disk request already incurs a latency on the order of 10 ms.

3.1. Data Layout in RAMA

Files in RAMA are composed of file blocks, just as in most file systems. However, RAMA uses reverse indices, rather than the direct indices used in most file systems, to locate individual blocks. It does this by grouping file blocks into disk lines and maintaining a per-line list of the file blocks stored there. Since a single disk line is 1 - 8 MB long, each disk in RAMA may hold many disk lines. A disk line, shown in Figure 1, consists of hundreds of sequential disk blocks and the table of contents (called a line descriptor) that describes each data block in the disk line. The exact size of a disk line depends on two file system configuration parameters: the size of an individual disk block (8 KB for the studies in this paper) and the number of file blocks in each disk line. The number of blocks per disk line was not relevant for the simulations discussed in this paper, since they did not run long enough to fill a disk line.

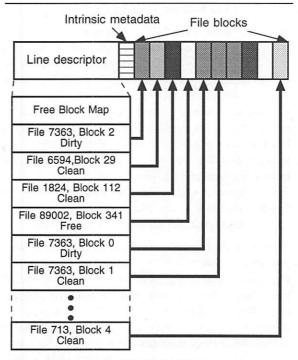


Figure 1: Layout of a disk line in RAMA.

The line descriptor contains a bitmap showing the free blocks in the disk line and a *block descriptor* for each block in the disk line. The block descriptor contains the identifier of the file that owns the block, its offset within the file, an access timestamp for the block, and

a few bits holding the block's state (free, dirty, or clean).

Every block of every file in RAMA may be stored in exactly one disk line; thus, the file system acts as a set-associative cache of tertiary storage. File blocks are mapped to disk lines using the function *diskline* = hash(*fileId*, *blockOffset*). This mapping may be performed by any node in the MPP without any file-specific information beyond the file identifier and offset for the block, allowing RAMA to be scaled to hundreds of processor-disk pairs.

The hash algorithm used to distribute data in RAMA must do two things. First, it must insure that data from a single file is spread evenly to each disk to insure good disk utilization. Second, it must attempt to map adjacent file blocks to the same line, allowing larger sequential disk transfers without intermediate seeks. This is done in RAMA by dividing the block offset by an additional hash function parameter s. This scheme yields the same hash value, and thus the same mapping from file block to disk line, for s sequential blocks in a single file. The optimal value for s depends on both disk characteristics and the workload [13]. S is set to 4 for the simulations in this paper.

While any node in the MPP can compute the disk line in which a file block is stored, direct operations on a disk line are only performed by the processor to which the line's disk is attached. This CPU scans the line descriptor to find a particular block within a disk line, and manages free space for the line. The remainder of the nodes in the MPP never know the exact disk block where a file block is stored; they can only compute the disk line that will hold the block. Since the exact placement of a file block is hidden from most of the file system, each node may manage (and even reorder) the data in the disk lines under its control without notifying other nodes in the file system.

RAMA's indexing method eliminates the need to store positional metadata such as block pointers in a central structure similar to a normal Unix inode. This decentralization of block pointer storage and block allocation allows multiple nodes to allocate blocks for different offsets in a single file without central coordination. Since there is no per-file bottleneck, the bandwidth RAMA can supply is proportional to the number of disks. If all nodes has the same number of disks, performance scales as the number of nodes increases.

The remainder of the information in a Unix inode—file permissions, file size, timestamps, etc.— is termed *intrinsic metadata* since it is associated with the file regardless of the media on which the file is

stored. The intrinsic metadata for a file is stored in the same disk line as the first block of a file in a manner similar to inodes allocated for cylinder groups in the Fast File System [12].

Since each block in RAMA may be accessed without consulting a central per-file index, the line descriptor must keep state information for every file block in the disk line. Blocks in RAMA may be in one of three states — free, dirty, or clean — as shown in Figure 1. Free blocks are those not part of any file, just as in a standard file system. Blocks belonging to a file are dirty unless an exact copy of the block exists on tertiary storage, in which case the block is clean. A clean block may be reallocated to a different file if additional free space is needed, since the data in the block may be recovered from tertiary storage if necessary.

RAMA, like any other file system, will fill with dirty blocks unless blocks are somehow freed. Conventional file systems have only one way of creating additional free blocks - deleting files. However, mass storage systems such as RASH [6] allow the migration of data from disk to tertiary storage, thus freeing the blocks used by the migrated files. RAMA uses this strategy to generate free space, improving on it by not deleting migrated files until their space is actually needed. Instead, the blocks in these files are marked clean, and are available when necessary for reallocation. RAMA also supports partial file migration, keeping only part of a file on disk after a copy of the file is on tape. This facility is useful, for example, for quickly scanning the first block of large files for desired characteristics without transferring an entire gigabyte-long file from tape.

3.2. RAMA Operation

A read or write request in RAMA involves only two nodes: the node making the request (the client) and the node on whose disk the requested data block is stored (the server). If several clients read from the same file, they do not need to synchronize their activities unless they are actually reading the same block. In this way, many nodes can share the large files used by parallel applications without a per-file bottleneck.

Figure 2 shows the flow of control and data for a file block read in RAMA; a similar sequence is used to write a file block. First, the client hashes the file identifier and offset, computing the disk line in which the desired block is stored. The client then looks up the owner of the disk line, and sends a message to that server. The server reads the line descriptor (if it is not already cached) and then reads or writes the desired block. If the operation is a write, the data to write goes

with the first message. Otherwise, the data is returned after it is read off disk.

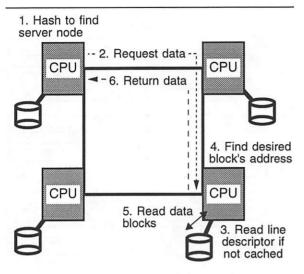


Figure 2: Steps required to read a file block in RAMA.

The common case for the file system is that the data is on disk. If a block not listed in the appropriate line descriptor is written, a free block is allocated to hold the data. If a read request cannot be satisfied, the block must exist on tertiary storage; a request is sent to the user-level tertiary storage manager process and the requested data is fetched from tape. While running the tertiary storage manager at user level adds context switch delays to the I/O time, the penalty of a few milliseconds pales in comparison to the tens of seconds necessary to fetch data from tape. Additionally, keeping tertiary storage management at user level allows much greater flexibility, as RAMA need not be recompiled (or even restarted) if a new tertiary storage device or migration policy is added.

4. Simulation Methodology

We used a simulator to compare RAMA's performance to that of a striped file system. The simulator modeled the pseudo-random placement on which RAMA is based, but did not deal with unusual conditions such as full disk lines. This limitation does not affect the findings reported later, since none of the workloads used enough data to fill the file system's disks. The simulator also modeled a simple striped file system using the same disk models, allowing a fair comparison between striping and pseudo-random distribution.

The interconnection network and disks in the MPP were both modeled in the simulator. While it would have been possible to model the applications' use of the network, this was not done for two reasons. First,

modeling every network message sent by the application would have slowed down simulation by a factor of 10 or more. Second, the network was not the bottleneck for either file system, as Section 5.3 will show. The simulator did model network communications initiated by the file system, including control requests from one node to another and file blocks transferred between processors. This allowed us to gauge the effect of network latencies on overall file system performance.

The disks modeled in the simulator are based on 3.5" low-profile Seagate ST31200N drives. Each disk has a 1 GB capacity and a sustained transfer rate of 3.5 MB/s, with an average seek of 10 ms. The seek time curve used in the simulation was based on an equation from [10] using the manufacturer's seek time specifications as inputs.

The workload supplied to the simulated file systems consisted of both synthetic benchmarks and real applications. The synthetic access patterns all transfer a whole file to or from disk using different, but regular, orderings and delays between requests. For example, one simple pattern might require each of *n* nodes to sequentially read 1/*n*th of the entire file in 1 MB chunks, delaying 1 second between each chunk. This workload generated access patterns analogous to roworder and column-order transfers of a full matrix.

Real access patterns, on the other hand, were generated by simulating the file system calls from a parallel application. All of the computation for the program was converted into simulator delays, leaving just the main loops and the file system calls. This allowed the simulator to model applications that would take hours to run on a large MPP and days to run on a workstation, and require gigabytes of memory to complete.

The modeled program used for many of the simulations reported in this paper was out-of-core matrix decomposition [5,19], which solves a set of n linear equations in n variables by converting a single $n \times n$ matrix into a product of two matrices: one upper- and one lower- triangular. Since large matrices do not fit into memory, the file system must be used to store intermediate results. For example, a 128K × 128K matrix of double-precision complex numbers requires 256 GB of memory — more than most MPPs provide. The algorithm used to solve this problem stores the matrix on disk in segments - vertical slices of the matrix each composed of several columns. The program processes only one segment at a time, reducing the amount of memory needed to solve the problem. Before "solving" a segment, the algorithm must update a it with all of the elements to its left in the

upper-triangular result. This requires the transfer of $c^2/2$ segments to decompose a matrix broken into c segments. The application prefetches segments to hide much of the file system latency; thus, the file system need only provide sufficiently fast I/O to prevent the program from becoming I/O bound. The point at which this occurs depends on the file system and several other factors — the number and speed of the processors in the MPP, and the amount of memory the decomposition is allowed to use.

5. Performance Comparison of RAMA and Striping

The RAMA file system is the product of a new of thinking about how a parallel file system should work. It is easier to use than other parallel file systems, since it does not require users to provide data layout hints. If this convenience came at a high performance cost, however, it would not be useful; this is not the case. I/O-intensive applications using RAMA may run 10% slower than they would if data were laid out optimally in a striped file system. However, improper data layout in a striped file system can lead to a 400% increase in execution time, a hazard eliminated by RAMA.

Parallel file system performance can be gauged by metrics other than application performance. Most parallel file systems use the MPP interconnection network to move data between processors. This network is also used by parallel applications; thus, a file system must that places a high load on the network links may delay the application's messages, reducing overall performance. Uniform disk utilization is another important criterion for parallel file systems with hundreds of disks. Using asynchronous I/O enables applications to hide some of the performance penalties from poorly distributed disk requests. However, uneven request distribution will result in lower performance gains from faster CPUs as some disks remain idle while others run at full bandwidth. RAMA meets or exceeds striped file systems in both network utilization and uniformity of disk usage.

5.1. Application Execution Time

The bottom line in any comparison of file systems is application performance. We simulated the performance of several I/O intensive applications, both synthetic and real, under both RAMA and standard striped MPP file system with varying stripe sizes. We found that RAMA's pseudo-random distribution imposed a small penalty relative to the best data

layout in a parallel file system, while providing a large advantage over poor data layouts.

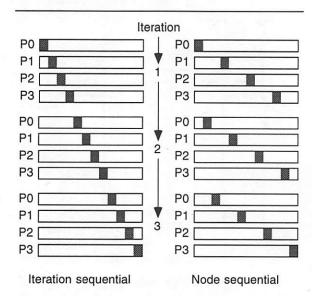


Figure 3: Transfer ordering for iteration sequential and node sequential access patterns.

The first benchmark we simulated read an entire 32 GB file on an MPP with 64 nodes and 64 disks. Each processor in the MPP repeatedly read 1 MB, waiting for all of the other processors to read their chunk of data before proceeding to the next one. These reads could be performed in two different orders resembling those shown in Figure 3: node sequential and iteration sequential. For node sequential access, the file was divided into 512 MB chunks, each of which was sequentially read by a single node. Iteration sequential accesses, on the other hand, read a contiguous chunk of 64 MB each iteration, using all 64 nodes to issue the requests. While the entire MPP appeared to read the file sequentially, each node did not itself issue sequential file requests.

We simulated this access pattern on several different configurations for the striped file system as well as the RAMA file system. Each curve in Figure 4 shows the time required to read the file for the striping configuration with N disked nodes and D disks per node, denoted by Nn, Dd on the graph. We varied the size of the stripe on each disked node to model different layouts of file data on disk. The horizontal axis of Figure 4 gives the amount of file data stored across all of the disks attached to a single disked node before proceeding to the next disked node in the file system. The dashed lines show the execution time for the iteration sequential access pattern, while the solid lines graph node sequential access. RAMA's performance for a given access pattern is constant since it does not

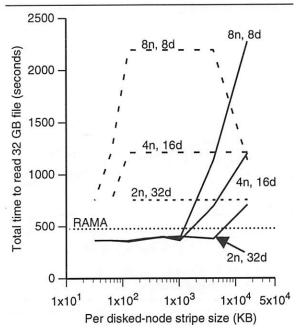


Figure 4: Time required to read a 32 GB file on an MPP with 64 processors and 64 disks.

use layout information; thus, there is only one execution time for each access pattern. Since the execution times for the different access patterns under RAMA were within 0.1%, RAMA's performance is shown as a single line.

As Figure 4 shows, RAMA is within 10% of the performance of a striped file system with optimal data layout. Non-optimal striping, on the other hand, can increase the time needed to read the file by a factor of four. Worse, there is no striped data layout for which both access patterns perform better than RAMA. There is thus no "best guess" the file system can make that will provide good performance for both access patterns. With RAMA, however, pseudo-random data layout probabilistically guarantees near-optimal execution time.

Real applications such as the out-of-core matrix decomposition described in Section 4, exhibit similar performance variations for different data layouts in a striped file system. As with the earlier benchmarks, however, RAMA provides consistent run times despite variations in the algorithm.

Matrix decomposition stresses striped file systems by only transferring a portion of the file each iteration. If each of these partial transfers is distributed evenly to all of the disks, the performance shown in Figure 5 results. Most of the data layouts for the striped file system allow the application to run without I/O delay, while only the largest file system stripe sizes are sub-

optimal. Performance under RAMA matches that of the best striped arrangements, and is better than execution time for the worst data layouts.

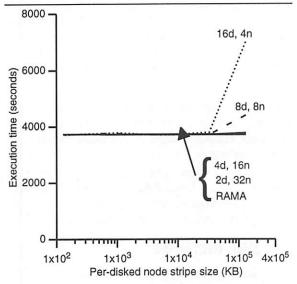


Figure 5: Execution time for LU decomposition under RAMA and striped file systems on a 64 node MPP with 64 disks.

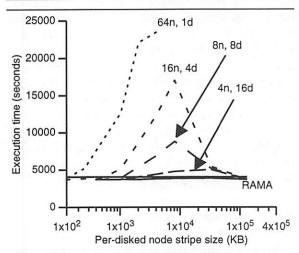


Figure 6: Execution time for LU decomposition with an alternate data layout.

Just a small change in the algorithm's source code governing the placement of data, however, can cause a large difference in performance for matrix decomposition under striping. The data in Figure 6 were gathered from a simulation of a matrix decomposition code nearly identical to that whose performance is shown in Figure 5. The sole difference between the two is a single line of code determining the start of each segment in the matrix. An minor arbitrary choice such as this should not result in a radical difference in performance; it is just this sort of dependency that

makes programming parallel machines difficult. Performance under file striping, however, is very different for the two data layouts. The small stripe sizes that did well in the first case now perform poorly with the alternate data layout. On the other hand, large stripe sizes serve the second case well, in contrast to their poor performance with the first data layout. In contrast, the execution times for the two variants using RAMA are within 0.1%.

Simulation results from other synthetic reference patterns and application skeletons showed similar results. A global climate model, for example, attained its highest performance for medium-sized stripes. Execution time using either large or small stripes was two to four times longer. Using RAMA's pseudo-random distribution, the climate model was able to run at the same speed as the optimal striped data layout.

5.2. Disk Utilization

There are two reasons for the wide variation in program performance under file striping: poor spatial distribution and poor temporal distribution of requests to disks. The first problem occurs when some disks in the file system must handle more requests than others because the application needs the data on them more frequently. Even if all of the disks satisfy the same number of requests during the course of the application's execution, however, the second problem may remain. At any given time, the program may only need data on a subset of the disks; the remaining disks are idle, reducing the maximum available file system bandwidth. RAMA solves both of these problems by scattering data to disks pseudo-randomly, eliminating the dissonance caused by conflicting regularity in the file system and the application's data layouts.

Figure 7 shows the average bandwidth provided by each of 64 disks in a striped file system during the decomposition of a $32K \times 32K$ matrix. The bandwidth is generally highest for the lowest-number disks because they store the upper triangular portions of each segment which are read to update the current segment. The disks on the right, however, store the lower triangular parts of the segments which are not used during segment updates. Since the file system is limited by the performance of the most heavily loaded disks, it may lose as much as half of its performance because of the disparity between the disks servicing the most and fewest requests.

To prevent this poor assignment of data to disks, RAMA's hash function randomly chooses a disk for each 32 KB chunk of the matrix file. The result is the distribution of requests to disks shown in Figure 8.

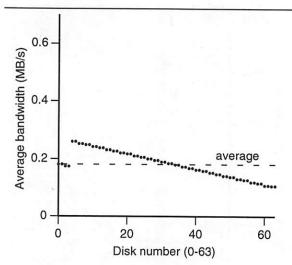


Figure 7: Disk utilization in a striped file system for a 32K × 32K matrix decomposition.

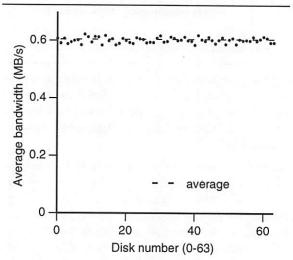


Figure 8: Disk utilization in RAMA for a $32K \times 32K$ matrix decomposition.

Each disk delivers between 0.584 and 0.622 MB/s for the 32K × 32K matrix decomposition, a spread of less than 6.5%. This difference is much smaller than the factor of two difference in the striped file system. Thus, RAMA can provide full-bandwidth file service to the application, while the striped file system cannot.

Striped file systems can also have difficulties with temporal distribution, as shown by the disk activity during a 1 GB file read graphed in Figure 9. The top graph shows the ideal situation in which every disk is active all of the time. Often, however, poor disk layout results in the bottom situation. Though every disk satisfies the same number of requests during the program's execution, only half of the disks are busy at

any instant, cutting overall bandwidth for the file system in half.

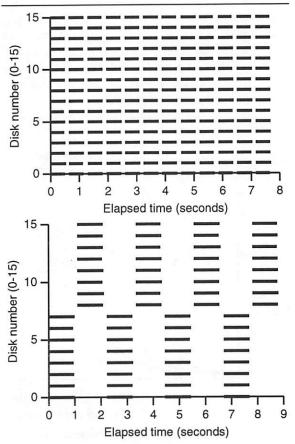


Figure 9: Disk activity over time for a striped file system during differently-ordered reads of a 1 GB file.

Here, too, RAMA's pseudo-random distribution avoids the problem. As Figure 10 shows, each disk is active most of the time. By randomly distributing the regular file offsets requested by the program, RAMA probabilistically assures that all disks will be approximately equally utilized at all times during a program's execution.

5.3. Interconnection Network Utilization

Many older parallel file systems [16,17] required data placement hints from programs to reduce network traffic as well as balance disk traffic. On some older machines, each interprocessor link was slower than 10 MB/s — hardly faster than a disk. On such a system, pseudo-random placement as done in RAMA would be a poor choice because it would place too high a load on the interprocessor links. However, interconnection networks have become faster; each processor in the Cray T3D [3] is connected to its

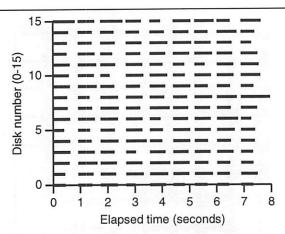


Figure 10: Disk activity over time for RAMA during a read of a 1 GB file. The access pattern is the same as the lower graph of Figure 9.

neighbors by six links each capable of transferring over 150 MB/s. The gap between network and disk speeds will only widen, since network technology is electronic while disk speeds are limited by mechanics.

As Figure 11 shows, the message traffic created by RAMA does not place high loads on a torus interconnection network with 100 MB/s links, even while all of the disks are transferring data at full speed. Average link utilization during the matrix decomposition ranged from 1.6% to 2.8%, leaving the remainder of the bandwidth for application-generated messages. The network load was evenly distributed throughout the matrix with no hot spots because the disks and requests to them were evenly spread. This uniform load decreases the travel time variation for "normal" messages, simplifying (a little bit) the creation of parallel programs.

The striped file system also caused relatively little congestion of the interconnection network — no link averaged more than 5.9% utilization over the course of the matrix decomposition. However, variation in file system link utilization was much higher than in RAMA. The links connecting to the disked nodes were, as expected, more heavily used by file system messages than those away from the disked nodes, as Figure 12 shows. The overall amount of file system message traffic was similar for RAMA and striping. However, RAMA's messages were more evenly spread through the interconnection torus. Thus, a side benefit for RAMA is better interconnection network load leveling from more uniform distribution of file system message traffic.

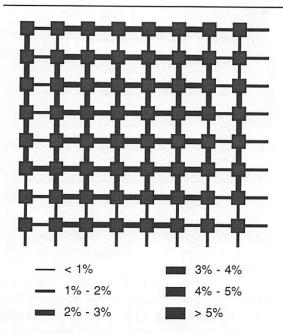


Figure 11: Interconnection network load under RAMA.

The shaded nodes have disks attached, all of which are being used to read a file at full speed.

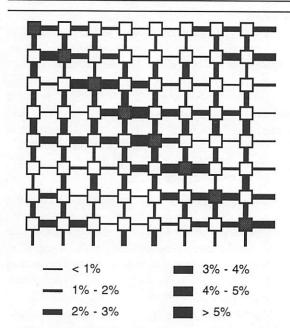


Figure 12: Interconnection network load under a striped file system. The program run is the same as in Figure 11.

6. Small File Performance

A major attraction of the RAMA file system is that it performs well on high-volume small file workloads as well as on supercomputer workloads. The workloads in Figure 13 use constant file sizes requested at different rates to generate each curve. Rather than use a closed system in which a fixed number of processes make requests as rapidly as possible, the RAMA simulator schedules requests according to a Poisson process whose average size and request rate are parameters to the workload. If there are too many outstanding requests, the simulator throttles the workload by delaying until an unfinished request completes, avoiding infinite queue growth.

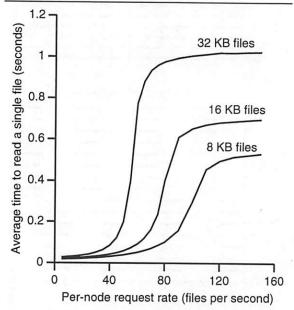


Figure 13: RAMA read performance for small files.

RAMA has low latency for small file transfers, enabling workstations connected to an MPP to access files directly instead of copying them to and from the MPP file system. Figure 13 shows RAMA's simulated performance on transfers of small files, 75% of which are reads. Even for 32 KB files, RAMA's performance does not begin to decline until the average request rate exceeds 40 requests per MPP node (and disk) per second. For the 16 × 8 processor mesh in Figure 13, this is an average rate of over 5000 requests per second.

RAMA is able to maintain this high level of performance for small files because file data is already distributed pseudo-randomly. The request stream from a workstation network results in a disk request stream similar to that for a single large file — both request lots of data in a somewhat random fashion. As with large files, no layout information need be supplied for small files, allowing workstations to use standard Unix file access semantics.

7. Future Work

The simulation results in this paper show that the RAMA file system design has great promise as a file system for future massively parallel machines. Many questions still remain to be answered, however. Issues to be explored further include RAMA's integration with tertiary storage and file migration, the testing of additional parallel applications, and the actual implementation of the RAMA file system using the experience gained from simulation.

One of the main attractions of RAMA for a scientific environment is its tight integration with tertiary storage. RAMA provides a facility unique among file systems for scientific storage — support for partial file migration. We will explore file migration algorithms, considering partial file migration and other developments in the fifteen years since [18].

We are also planning to build a prototype version of RAMA on a parallel processor. This can be done in two ways: as a software library layered over a generic file system, or as a replacement for an MPP file system. The first approach would be simpler, but the latter will prove a better test of RAMA's ideas. A true RAMA system will provide a good testbed for I/O-intensive parallel applications. Running real programs on this testbed will show that programmers need not spend their energy trying to lay data out on disk; the file system can do the job just as well using pseudorandom placement.

An implementation of RAMA will also be a good place to explore RAMA's design space. How much consecutive file data should be stored on a disk before randomly selecting another? How big should a disk line be? How will performance be affected as disk lines fill and allocation becomes more difficult? A RAMA prototype will allow us to address these issues by experimenting on a real system.

8. Conclusions

Traditional multiprocessor file systems use striping to provide good performance to massively parallel applications. However, they depend on the application to provide the file system with placement hints. In the absence of such hints, performance may degrade by a factor of four or more, depending on the interaction between the program's data layout and the file system's striping.

RAMA avoids the performance degradation of poorly configured striped file systems by using pseudorandom distribution. Under this scheme, an application is unlikely to create hot spots on disk or in the network because the data is not stored in an orderly fashion. Laying files on disk pseudo-randomly costs, at most, 10-20% of overall performance when compared to applications that stripe data optimally. However, optimal data striping can be difficult to achieve. Applications using striped file systems may increase their execution time by a factor of four if they choose a poor data layout. This choice need not be the fault of the programmer, as simply using a machine with its disks configured differently can cause an application's I/O to run much less efficiently. RAMA's performance, on the other hand, varies little for different data layouts in full-speed file transfers, matrix decomposition, and other parallel codes.

The flexibility that RAMA provides does not exact a high price in multiprocessor hardware, however. RAMA allows MPP designers to use inexpensive commodity disks and the high-speed interconnection network that most MPPs already have. It is designed to run on an MPP built from replicated units of processor-memory-disk, rather than the traditional processor-memory units. This method of building MPPs removes the need for a very high bandwidth link between an MPP and its disks; instead, the file system uses the high-speed network that already exists in a multiprocessor. Since the file system is disk-limited, though, the network is never heavily loaded.

Disks, too, are utilized well in RAMA. Pseudorandom distribution insures an even distribution of data to disks. Disk requests are evenly distributed to disks in time as well as in space. Thus, no disk serves as a bottleneck by servicing too many requests at any time. In addition, all disks are used nearly equally at every step of an I/O-intensive application without the need for data placement hints.

The simulations of both synthetic traces and cores of real applications show that the pseudo-random data distribution used in RAMA provides good performance while eliminating dependence on user configuration. While RAMA's performance may be 10-15% lower than an optimally configured striped file system, it provides a factor of four or more performance improvement over a striped file system with a poor layout. It is this portability and scalability that make RAMA an excellent file system choice for the multiprocessors of the future.

References

[1] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur. "PASSION: Parallel And Scalable Software for Input-Output." Technical Report

- NPAC Technical Report SCCS-636, NPAC and CASE Center, Syracuse University, Sept. 1994.
- [2] P. F. Corbett, D. G. Feitelson, J.-P. Prost, and S. J. Baylor. "Parallel access to files in the Vesta file system." In *Proceedings of Supercomputing* '93, pages 472–483, Portland, Oregon, Nov. 1993.
- [3] Cray Research, Inc. "Cray T3D system architecture overview manual," Sept. 1993. Publication number HR-04033.
- [4] P. Dibble, M. Scott, and C. Ellis. "Bridge: A high-performance file system for parallel processors." In *Proceedings of the Eighth International Conference on Distributed Computer Systems*, pages 154–161, June 1988.
- [5] G. A. Geist and C. H. Romine. "LU factorization algorithms on distributed-memory multiprocessor architectures." SIAM Journal of Scientific and Statistical Computing, 9(4):639–649, July 1988.
- [6] R. L. Henderson and A. Poston. "MSS-II and RASH: A mainframe UNIX based mass storage system with a rapid access storage hierarchy file management system." In USENIX — Winter 1989, pages 65–84, 1989.
- [7] D. W. Jensen and D. A. Reed. "File archive activity in a supercomputer environment." Technical Report UIUCDCS-R-91-1672, University of Illinois at Urbana-Champaign, Apr. 1991.
- [8] R. H. Katz, G. A. Gibson, and D. A. Patterson. "Disk system architectures for high performance computing." *Proceedings of the IEEE*, 77(12):1842–1858, Dec. 1989.
- [9] D. Kotz. "Disk-directed I/O for MIMD multiprocessors." Technical Report PCS-TR94-226, Dept. of Computer Science, Dartmouth College, July 1994.
- [10] E. K. Lee and R. H. Katz. "An analytic performance model of disk arrays." In *Proceedings of SIGMETRICS*, pages 98–109, May 1993.
- [11] S. J. LoVerso, M. Isman, A. Nanopoulos, W. Nesheim, E. D. Milne, and R. Wheeler. "sfs: A parallel file system for the CM-5." In Proceedings of the 1993 Summer Usenix Conference, pages 291–305, 1993.
- [12] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. "A fast file system for UNIX." ACM Transactions on Computer Systems, 2(3):181– 197, Aug. 1984.

- [13] E. L. Miller. Storage Hierarchy Management for Scientific Computing. PhD thesis, University of California at Berkeley, to be published in early 1995.
- [14] E. L. Miller and R. H. Katz. "Input/output behavior of supercomputing applications." In *Proceedings of Supercomputing '91*, pages 567–576, Nov. 1991.
- [15] E. L. Miller and R. H. Katz. "An analysis of file migration in a Unix supercomputing environment." In *USENIX—Winter 1993*, pages 421– 434, Jan. 1993.
- [16] P. Pierce. "A concurrent file system for a highly parallel mass storage system." In Fourth Conference on Hypercube Concurrent Computers and Applications, pages 155–160, 1989.
- [17] T. W. Pratt, J. C. French, P. M. Dickens, and S. A. Janet, Jr. "A comparison of the architecture and performance of two parallel file systems." In Fourth Conference on Hypercube Concurrent Computers and Applications, pages 161–166, 1989.
- [18] A. J. Smith. "Long term file migration: Development and evaluation of algorithms." *Communications of the ACM*, 24(8):521–532, August 1981.
- [19] D. Womble, D. Greenberg, S. Wheat, and R. Riesen. "Beyond core: Making parallel computer I/O practical." In *Proceedings of the 1993 DAGS/PC Symposium*, pages 56–63, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

Author Information

Ethan Miller is an assistant professor at the University of Maryland Baltimore County. He received a ScB from Brown in 1987 and an MS from Berkeley in 1990. He will complete his PhD at Berkeley in early 1995. His research interests are file systems and data storage for high performance computing, including both disk and tertiary storage. Surface mail sent to the Computer Science Department, UMBC, 5401 Wilkens Avenue, Baltimore, MD 21228 will reach him, as will electronic mail sent to elm@cs.umbc.edu.

Randy Katz has been on the Berkeley faculty since 1983. He received his MS and PhD at Berkeley in 1978 and 1980 respectively. He received his AB degree from Cornell University in 1976. He may be contacted by mail at the Computer Science Division, University of California, Berkeley, CA 94720, and by electronic mail at randy@cs.berkeley.edu.

POTPOURRI I

Session Chair: Greg Minshall, Novell, Inc.

NOTES

Implementing Real Time Packet Forwarding Policies using Streams

Ian Wakeman, Atanu Ghosh, Jon Crowcroft *
Computer Science Dept, University College London,
Gower Street, London WC1E 6BT.

Van Jacobson and Sally Floyd
Lawrence Berkeley Laboratory
One Cyclotron Road
Berkeley, CA 94720

November 14, 1994

Abstract

This paper describes an implementation of the class based queueing (CBQ) mechanisms proposed by Sally Floyd and Van Jacobson [1] [2] to provide real time policies for packet forwarding. CBQ allows the traffic flows sharing a data link to be guaranteed a share of the bandwidth when the link is congested, yet allows flexible sharing of the unused bandwidth when the link is unloaded. In addition, CBQ provides mechanisms which give flows requiring low delay priority over other flows. In this way, links can be shared by multiple flows yet still meet the policy and Quality of Service (QoS) requirements of the flows.

We present a brief description of the implementation and some preliminary performance measurements. The problems of packet classification are addressed in a flexible and extensible, yet efficient manner, and whilst the Streams implementation cannot cope with very high speed interfaces, it can cope with the serial link speeds that are likely to be loaded.

1 Introduction

The Internet is fast approaching a period of revolutionary change in the services provided and how they are paid for. New architectures and protocols are being designed and imple-

mented to extend the Internet to support Integrated Services, based on the work of the INT-SERV working group of the IETF [3]. It is envisioned that audio, video and other realtime services will be sent over the Internet. The ongoing commercialisation of the Internet necessitates a new model of service, where the users and providers exchange money for a guarantee of a basic level of service. Since the delivered service for both real time applications and contractual guarantees is dependent upon the mix of packets on the links and in the switches, a key component of the new Internet will be the packet forwarding scheduler within the switch. This must provide both a method for sharing bandwidth amongst the organisations who pay for the link and provide appropriate levels of Quality of Service for flows with real-time requirements.

One vision of how to design this building block has been offered by Sally Floyd and Van Jacobson [1] [2]. They start from the premise of link sharing, where links are leased by multiple organisations, or agencies, who then require a guarantee of a share of the bandwidth when they need it, but if the bandwidth is not used, then other users can send packets. These requirements can only be satisfied in any sensible manner through the scheduling of packets to be forwarded. Each of the agencies are guaranteed a minimum amount of the bandwidth, with the proviso that any instantaneously unused bandwidth is shared amongst the agencies in some previously agreed upon manner. This technique for allocating band-

^{*}Supported by DARPA grant number AFOSR890514 and generous donations from Sun Microsystems Laboratories Inc

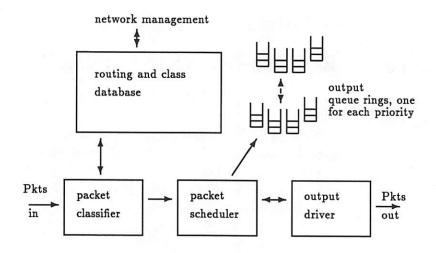


Figure 1: Conceptual Breakdown of the CBQ filter Code

width can be naturally extended to provide the allocation mechanisms for real-time traffic, providing the required Quality of Service (QoS) through guaranteeing bandwidth and low delay. The unifying abstraction for linksharing and real-time traffic is the use of a class hierarchy; each class is provided with a share of the bandwidth and a delay characteristic, and the hierarchy enables mechanisms and policies for "borrowing" of bandwidth from other classes.

The conceptual breakdown of how CBQ should be implemented has been proposed by Van Jacobson and is illustrated in Figure 1. The classifier interprets the header information of an incoming packet to determine the class of service that the packet should receive from the scheduler. The classifier returns a pointer to the class structure that holds the queues and associated information. The packet is placed on the appropriate queue by the scheduler if the queue is not full. The output driver works asynchronously, invoking the scheduler to determine from which queue to send the next packet, according to the current utilisation of the queues and their priorities.

Classifying a packet is very similar to the problem of determining the route matching a destination address in a packet. In both cases, fields in the header are used to look up information in a table. However, the classification problem is more complex because the patterns upon which the packet may match a class must be more general and can match any part of

the header. For instance packets may be from one of multiple agencies, requiring the examination of destination and source addresses, classified on transport or other protocols such as TCP, UDP or ICMP, or applications such as ftp or telnet, requiring the examination of the port numbers. For video streams we may look even further within the packet to determine the level and "droppability" of a packet within a hierarchically encoded video stream [4].

It is a point of some controversy as to how widely the above mechanisms need to be fielded within the Internet. It could be argued that they need only be used where the links are heavily utilised, and thus subject to congestion. Links which have low utilisation can supply the necessary Quality of Service for all types of stream with normal FIFO packet scheduling, since queues on the links are very small or non-existent. The motivation for the work described in this paper came from attempting to make the allocation of bandwidth more efficient on the Trans-Atlantic link (known as the FAT pipe) used to connect the data networks of the UK Ministry of Defence (MoD) and the Department of Defence (DoD). of the European Space Agency (ESA) and NASA, and of the UK academic IP network and the US academic network. Currently, the link is hard-multiplexed into three parts, one for each agency pair1. However, large cost sav-

¹Currently split into an E1 and a T1 link, with the T1 link split into two equal channels. For details on

ings could be achieved on an expensive link if we guarantee each of the agencies a minimum share of the bandwidth for their mission, yet could allow any unused bandwidth to be used by agencies with excess traffic. The current version of the CBQ filter is fielded on the FAT pipe.

In this paper we describe the implementation of a programmable CBQ mechanism within the Streams implementation of IP forwarding. The classes and the patterns that determine the membership of a particular class are compiled in user space, and lookup engines are chosen to optimise the per-packet lookup code. The classes are then downloaded to the CBQ filter module, which schedules the packets using the downloaded information. The scheduling code is largely based upon the work of Lawrence Berkeley Laboratory (LBL), whilst the classifier and the surrounding infrastructure are the work of University College London (UCL). The particular choices made in the design of the classes and the classifier are described in Section 2, the design of the Streams module and its virtual interface are described in Section 3 and performance measurements reported in Section 4. We conclude in Section 5 with ruminations on the feasibility of this design and the limitations of the design from the Streams mechanism and the pointers to future work.

2 Link Sharing **Policies** and the Classifier

The problem is to map the high level specification of policies onto a classifier that maps a packet into a particular class so that the necessary information can be found for scheduling the packet for output.

The high level specification of policies should be at a level of abstraction that can be used in the negotiation of legally binding sharing of the link. For instance, the agencies should be named as NASA or UK academic organisations, packet types should be specified by the services they provide - interactive data, bulk data, video data, audio data, specific sites, protocol suites (DECnet, IP etc.), and specific protocols.

Our initial class structure definition has the

following hierarchical ordering when we are allocating the shares of the bandwidth:-

- 1. Agency
- 2. Protocol Suite
- 3. Protocol
- 4. Service

This partitioning first allows the bandwidth to be divided up by organisation, allowing arbitrary levels in this level. The next level is a suggestion that bandwidth should be given to protocol suites separately, so, for instance, OSI and IP packets are separated out in treatment. However, an actual implementation would reverse the process in classifying the packets, since the agency could only be discovered after knowing the structure of the packet. In this paper, we consider only IP. The partitioning by protocols allows protection against various forms of link sharing at the congestion control level - e.g. none vs slow start [6] vs responsible second order sharing [7] [8], [9] and to specify appropriate actions when the classes are exceeding their bandwidth allocation when the link is loaded. The final division by service allows us to divide up bandwidth amongst the applications. By adding a priority level at this point and implementing sensible borrowing policies in the scheduler, we can ensure that when there is congestion the service offered to the applications is degraded according to their importance, such as video packets being dropped before audio in a video conference.

2.1 What's a class?

Having derived the abstractions used to map a packet onto a class, we then want to consider the actual attributes that will be attached to a class. A class is :-

- A share of the bandwidth
- A priority
- A parent class
- A set of pattern tuples over a packet, P_j = $\{\{A_i, V_i\}\}$, where a pattern is:-
 - A pattern A:.
 - A mask of significant bits V_i.

the usage see [5]

As happens in the solution to many computing problems, we define a hierarchy of classes. Rather than attempting to map a packet onto a class in one step, our pattern matching proceeds in order down the tree. We thus compare the packet against the patterns of the children of the current class. To ensure that a packet maps unambiguously only onto a single class, the patterns specified should follow the following constraint in the general case:-

• For each child of a given parent, each pattern of each child should be distinguishable from the patterns of all other children, ie for children i and j, there should exist no pattern tuples in i and j such that $A_i \wedge V_i \equiv A_j \wedge V_j$.

To simplify the implementation, and because the mapping abstractions correspond nicely to the packet headers we are using, the patterns are further constrained to map directly onto fields in the protocol header. For our initial IP implementation, the agency maps onto the destination and source addresses2, the protocol maps onto the protocol field, whilst the application maps onto the port numbers in UDP or TCP [10,10]. The abstraction of a class and the tree structure of the classes leads to the number of patterns we have to compare a packet against being exponential in the depth of the tree, using Figure 2. In addition, the number of patterns at a particular level may be very large - e.g. at the agency level, the current design of protocols insists that we compare against the destination and source net numbers of the composite networks of the agency. For the initial case this can be as large as 10,000 or more (UK and US academic institutions). Furthermore, since the patterns will be repeated at each level (because each agency will want to partition traffic along similar lines), this will require a large number of patterns to be constructed if we attempt to classify the pattern in one step. Making a comparison at each level of the tree reduces the total number of patterns that need to be stored, but increases the number of comparison operations.

However, this is not a problem, since the class tree will be shallow - generally three or

four levels - and we can select a lookup engine for a given class level according to the nature of the patterns to be compared against. The use of IP and simple classes allows the classification to proceed by a sequence of hash table or simple table lookups. Note that there is a default class as a child of the root of the tree into which a packet falls if it does not map to any other class. This default class gives the lowest quality of service, since it is not being used by any of the agencies paying for the link. It should be noted that the existence of a default class will encourage agencies to offer their link as a transit link for other agencies, and promote greater connectivity.

The use of sequenced lookups for the classes thus adds additional parameters to the class a lookup engine function, and the data structures for the lookup engine.

Its then becomes natural for the network manager to specify the full set of parameters for the classes in a file, which is then compiled. Simple parenthesised definitions are used to define the tree. Patterns are defined very simply at present, just as a specification of the field within the IP packet and the pattern to compare it against.

The classifier compiler types the set of patterns that define the child classes of a class, checking that they can all be used to create a single engine. It then creates the data structures for the engine, placing the engine type in the class data, along with functions to lookup, insert and relocate the data, along with other parameters to describe a class.

Relocation of the data structures is necessary to allow the compiler to work in user space and then pass the data structures down to the kernel driver. The design is split in this manner so that we could experiment with reservation strategies - adjustments to the bandwidth and the priorities of classes can be worked out in user space and then passed down to the kernel. Additional management functionality, such as Management Information Base (MIB) creation etc., can be easily designed and added at a later date. An example configuration file is shown in Figure 3, which is used in the tests described below. An alternative approach to packet classification is to use a generalised patricia lookup engine [11], [12]. Patricia constructs a tree which is traversed by testing only against the bits of the key which differentiate the patterns stored

²If a member of agency A is temporarily on walkabout, yet still wishes to use the share of the link, an additional pattern can be added to the class to cope with the temporary addresses.

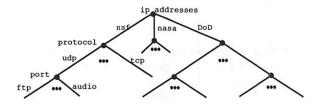


Figure 2: Example Class tree hierarchy

```
# bandwidth of wire in
# nanoseconds per byte
linkspeed 20000
{ name ucl-cs
# percentage of bandwidth
  bandwidth 90
  pattern [addr 128.16.0.0]
  { name ucl-cs-tcp
# percentage of total bandwidth
    bandwidth 50
# action when overlimit
    overlimit drop
    pattern [proto tcp]
    { name highTCP
# high number = high priority (max 7)
      priority 6
      bandwidth 29
      overlimit drop
      pattern [port 5001]
    { name lowTCP
      bandwidth 19
      overlimit drop
      pattern [port 5002]
 }
  { name ucl-cs-udp
    bandwidth 39
    overlimit drop
    pattern [proto udp]
    { name CBR
      priority 7
      bandwidth 38
      overlimit drop
      pattern [port 4579]
}
```

Figure 3: Classes used in Testing the Streams Module

in the tree. The technique has been generalised to cope with masked keys by Halpern [13] and Tsuchiya [14]. However, because our patterns may have no bits in common, such as matching on one of either destination or source address, multiple passes are still required through the tables. In addition, tests against a generalised Patricia tree lookup engine show the sequenced table lookup as more efficient (Section 4).

Another common mechanism for classifying packets is the automata used inside the Berkeley Packet Filter (BPF) and other similar entities [15]. These are designed to match a narrow range of patterns across a packet, and to do this very efficiently. However, the range of patterns we expect the classifier to handle is very wide, which would end up with a number of automata which would all need to be tried sequentially. Therefore we have generalised the framework of the classifier to use generalised engines. This does not preclude use of the a BPF automata as a particular instance of an engine, if that is the most suitable.

2.2 The scheduling properties of a Class

Each of the classes maintains an exponentially weighted moving average (EWMA) (avgidle in Figure 4) of the idle period between packets, updated on every packet using the time it would take to send the same size packet using the percentage of the bandwidth allocated to the class. When the average is less than zero, a class is transmitting more than its share of the bandwidth and is thus "overlimit". If the class of a packet is overlimit, the "borrow" class of the packet is examined to see if the original class can borrow bandwidth to transmit the packet, and so on up the hierarchy. If the class goes overlimit and can't borrow,

```
struct rm_class {
 mbuf_t *tail; /* tail of circularly linked output g */
 struct timeval last; /* time last packet sent */
 struct timeval undertime; /* time can next send */
                     /* != 0 if delaying */
  int sleeping;
  int qcnt; /* # packets in queue */
  int avgidle; /* EWMA of idle time between pkts */
  struct rm_class *peer; /* Linked list of same priority classes */
 rm_class *borrow; /* Class to borrow bandwidth from */
 struct rm_class *parent; /* Parent class */
 struct rm_ifdat *ifdat; /* Output Device data structure */
  int priority; /* Class priority */
  int maxidle; /* Roof of avgidle */
  int offtime; /* Penalty added to class when overlimit */
 int qmax; /* Maximum queued pkts */
  void (*overlimit)(); /* Action to take when we can't borrow */
```

Figure 4: The Class Structure related to Scheduling (after Sally Floyd)

then the overlimit action is invoked, such as dropping the packet, or delaying the packet. The maximum size of the avgidle estimator is limited by the maxidle parameter which prevents the class from building up credit when it isn't transmitting, and so limits the maximum burst size from the class.

The scheduling code maintains equal priority classes with traffic to send in circular queues. The packet to send is either from the highest priority class which is underlimit, or from the highest priority class which is overlimit but can borrow from classes above it in the borrowing hierarchy of classes. After a packet is sent from a particular class the queue pointer is advanced to the next position in the queue. Thus we implement prioritised round robin as long as classes are underlimit. When the class goes overlimit, it allows other underlimit classes to send first and so prevent starvation of any class. In this way, real time flows can ensure low delay by setting the priority of the class high and ensuring that the bandwidth share requested is sufficient such that they are always underlimit. When the queues are full or when the overlimit action specified is to drop a packet, the current implementation drops from the tail. However, other mechanisms such as dropping from a random position within the queue are possible [16]. The details of scheduler are described more fully in

[1].

3 Implementation of a Streams-based Packet Forwarding Engine

The Streams "plumbing" used for the CBQ filter can be seen in Figure 5. The CBQD module is used to communicate with the CBQ filter module and to download the class buffers and to obtain statistics on usage.

Our target scenario was for the workstation, a SparcClassic running Solaris 5.2, to sit transparently in front of the router whose output serial line we wished to protect3. The workstation and the router would be connected by Ethernet or some high bandwidth link. Thus we had to emulate the speed of the router interface in the CBQ software. This is done by tracking the time that the packet would be sent from the interface if it were the speed of the serial line in virtual time and suspending transmission if the virtual time gets too far in front of real time - ie the queue in the router builds up. We rely on incoming packets and a backstop timer to ensure that the transmission is continued at some point after being

³The obvious place to implement this code is in the router, but we had no way of modifying the router.

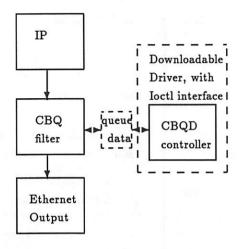


Figure 5: The Streams Plumbing of the CBQ filter modules.

suspended. An additional advantage of running in "virtual" time is that the code becomes independent of output completion interrupts. The original LBL code used the completion interrupt to service another packet, but device drivers in Solaris do not provide this interrupt.

Implementing the CBQ filter as a Streams module and driver had some excellent benefits. We could introduce code into the protocol stack without requiring changes to the kernel source, with one major proviso. At the time the work was started the Streams plumbing code was built into a program ("ifconfig"); it was therefore necessary to take a copy of the "ifconfig" program and modify the source to enable us to insert the CBQ module at the correct point in the Stream. It would have been more convenient if the Streams implementations had used a configuration file to set up the plumbing, as some other implementations do.

As Solaris only supports dynamically loadable drivers we had hoped that it would be possible to write and debug the CBQ module without ever having to reboot the development machine. Unfortunately there was no obvious way of tearing down a Stream and creating a new one with a new CBQ module. Therefore it was necessary to reboot each time we wished to test a new module. If there were any serious bugs in the CBQ module the only way of recovering was to boot the machine from the boot prompt and edit the relevant startup files to not use the CBQ module. One way we attempted to circumvent this problem was to

try and create a file before attempting to load the CBQ module - if this file already existed then the CBQ module would not be placed in the Streams stack. Unfortunately the network is configured so early in the Solaris boot sequence that the filestores are still readonly, so this strategy failed.

The scheduling code used was written at LBL and had been written for a BSD derived kernel. The first task was to change the code to use the Streams interface. This turned out to be quite easy; the BSD interface structure mapped quite simply onto a Streams queue and the "mbuf" structures mapped onto the Streams message buffers with the aid of some macros. The major change was the addition of some locks. The classification code was then integrated into the CBQ module.

A mechanism was required to control the CBQ module, to enable/disable the classification and to change the tables. The standard mechanism for sending information to drivers/modules is by obtaining a file descriptor to the driver/module and then issuing an "ioctl" which is understood by the driver. However the CBQ module is below a Streams multiplexor, and there is no way that the multiplexor layer can correctly deliver the ioctl message without rewriting the multiplexor code. The only way around this problem was to write a CBQ driver which accepts "ioctl" requests and makes a direct connection to the CBQ module, as in Figure 5, using a shared piece of memory in the queue structure.

Code	Lookups	Total Time spent /s	Mean ms/Call
UCL Classifier	15186	0.20	0.01
Patricia	15186	0.47	0.03

Table 1: Classifier test results

4 Performance measurements

4.1 Classifier performance

The classifier is designed to be flexible in the lookup engines used on the packets. The alternative design choice would have been to use a single engine, such as the modified Patricia code from Joel Halpern [13]. We compared the performance of the two approaches, using a classifier with multiple hash and normal table lookups with a modified Patricia engine. We specified the classes to consist of 1920 addresses culled from the NSFNet acceptable use database, with child classes of UDP and TCP traffic, and child classes of Telnet and FTPdata for the TCP class. We profiled the engines using the gprof profiler on a sparcStation 10 under Solaris 2.3, and in all tests, the UCL classifier code was faster, even though the patricia code ignored the destination address and port fields, due to the limitations described in Section 2. We present sample results from a test designed to exercise all paths of the engine in Table 1.

4.2 Streams Module Performance

In the experiments described below, we use the setup described in Figure 7. Both interfaces of the CBQ filter machine are on the same Ethernet, so we can see both input and output packets using a single tcpdump [17].

To measure the throughput of the Streams module, we measured the time taken to forward a series of packets well below the specified output rate of the filter for one of the connections on its own, and compared these with the performance of IP forwarding without the CBQ module. The results can be seen in Figure 6. These suggest that a sparcClassic can forward 700 Kbit/s of minimum sized packets, or 7 Mbit/s of maximum sized packets on an Ethernet, and that traversing the CBQ

module takes in the order of 300-450 microseconds. It should be noted that the current code is unoptimised - it does an extra copy to ensure alignment of the headers, and will pull up buffers without hesitation. The next version of the code will remove these inefficiencies.

To illustrate the performance of the Streams code in separating real packet streams, we set up an experiment using the classes in Figure 3. The three flows are routed into the CBQ filter which exists on a dual-homed host with both ethernet outputs on the same physical segments. The flows are then routed to their target hosts via an ordinary cisco router. The return path does not pass through the CBQ filter. The medium priority TCP flow starts first, followed twenty seconds later by the low priority TCP flow. After a further twenty seconds, we get a minute of high priority constant bit rate UDP packets. The constant bit rate traffic is intended to simulate audio traffic.

The data gathered from the tcpdump were analysed and can be seen in Figure 8. The data are clumped in five second buckets for clarity. It is important to realise that the constant bit rate stream suffered no loss, whilst the TCP streams incurred loss when the congestion window was opened too large. In addition, the streams are kept to within some margin of their allocated shares of the 62.5 KByte/s that we have set the virtual Interface to

The time series illustrating the delays suffered by the packets, measured by the time difference between the IP packets entering the filter and emerging on the wire again can be seen in Figure 9. The data are bucketed in 1 second buckets. The high priority CBR stream suffers low delay, whilst the TCP streams suffer variable delays, dependent upon whether they are underlimit and thus transmitted in priority order, or overlimit and thus transmitted after any underlimit classes with traffic to send have been sent.

Non-intuitively the average delay experienced by the higher priority TCP class is larger

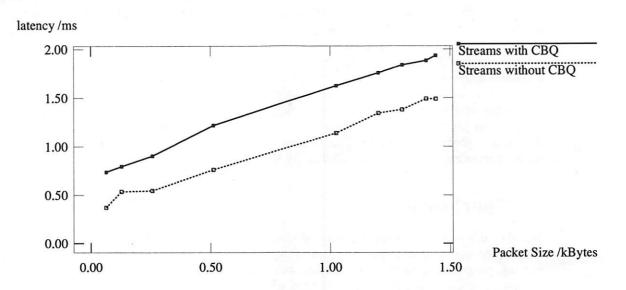


Figure 6: Latency of Streams module against size of packet

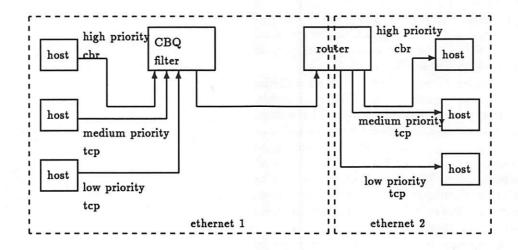


Figure 7: Experimental Setup for Testing the Streams Module

than the average delay of the lower priority TCP class. This is an artifact of the particular borrowing strategy we've implemented. Packets that are sent in the higher priority TCP class when this class is underlimit are at higher priority and thus suffer low delay. When this class is overlimit, it has greater claim on unused bandwidth than any other overlimit class, but at lower priority than any underlimit class, so the packets sent using this bandwidth have a higher delay, contributing to an overall higher delay $(de\bar{l}ay = (delay_{low} \times packets_{low} + delay_{high} \times packets_{high})/packets_{total})$. Alternative borrowing schemes are detailed in [1].

tricia derived algorithms, to the folks at University of London Computer Centre, SuraNet and BBN for their work and help in putting the CBQ code on the FATpipe and to Greg Minshall and the anonymous referees for their helpful comments.

5 Conclusion

We have described an implementation of the class based queueing strategy to provide link sharing, suggested by Sally Floyd and Van Jacobson. The Streams installation provided a modular environment in which to work. However, a number of factors proved a hindrance. The Solaris 2.3 release currently does not allow modules to be downloaded dynamically below the IP code. This slowed the development somewhat. In addition, some problems were encountered in determining when the packet had left the output driver.

The code is currently in use on part of the traffic passing over the UK-US FATpipe. This partitions traffic amongst a number of agencies, and allows the guaranteeing of multicast traffic for real-time applications a minimum of bandwidth and low delay scheduling.

We plan to extend this work to optimise the lookup engines and to allow more sophisticated pattern matching within the classifier. A remote management option will be added so that classes can be added, changed and deleted dynamically, without having to download all classes again. With this in place, we shall be able to experiment with real-time reservation, such as RSVP [18] and traffic management within the Internet.

An unsupported version of the code can be found on ftp://cs.ucl.ac.uk/darpa/cbq.tar.Z.

Acknowledgements

We are deeply indebted to Joel Halpern and Paul Francis for their code and help on Pa-

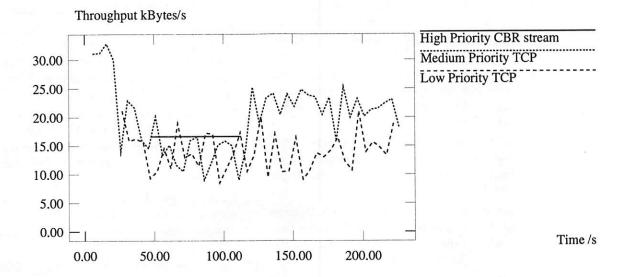


Figure 8: Throughput for the illustrative Streams Experiment

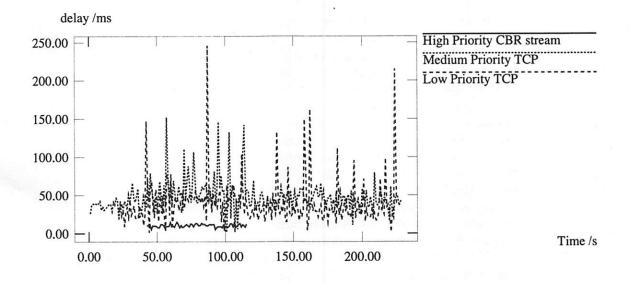


Figure 9: Delay for the illustrative Streams Experiment: Average delays - CBR stream - 0.01s, high priority TCP - 0.045s, low priority TCP - 0.038s

References

- Sally Floyd, "Link-sharing and Resource Management Models for Packet Networks," Submitted to ACM/IEEE Transactions on Networking.
- [2] Sally Floyd & Van Jacobson, "Class based queueing for policy based resource sharing," 1993, Internal presentation and private email.
- [3] R. Braden, D. Clark & S. Shenker, "Integrated Services in the Internet Architecture: an Overview," rfc1633 (September 1994).
- [4] Ian Wakeman, "Packetised Video: Options for interaction between the User, the Network and the Codec," The Computer Journal 36,1 (February 1993).
- [5] Ian Wakeman, Dave Lewis & Jon Crowcroft, "Traffic Analysis of trans-Atlantic traffic," Computer Communications 16 (June 1993), 376,388.
- [6] Van Jacobson, "Congestion Avoidance and Control," Proceedings ACM SIGCOMM Symposium (August 1988).
- [7] Ian Wakeman & Jon Crowcroft, "A Combined Admission and Congestion Control Scheme for Variable Bit Rate Video," To be published in Journal of Distributed Systems Engineering (October 1992).
- [8] Wang & Crowcroft, "A New Congestion Control Scheme: Slow Start and Search (Tri-S)," Computer Communications Review 21 (January 1991), 32-43.
- [9] Lawrence S. Brakmo, Sean W. O'Malley & Larry L. Peterson, "TCP Vegas: New techniques for congestion detection and avoidance," Proceedings of ACM Sigcomm94, London (September 1994).
- [10] D. Comer, Interworking with TCP/IP, Principles, Protocols and Architecture, Prentice Hall International, ISBN 0 13 468505 9, 1988.
- [11] Donald R. Morrison, "PATRICIA Practical Algorithm to Retrieve Information Coded In Alpha-numeric," Journal of the ACM 15,4 (October 1968).
- [12] Robert Sedgewick, Algorithms, Addison-Wesley, 1988.

- [13] Joel Halpern, "Modifications of Patricia Trees for Handling values with discontiguous masks with emphasis on internet routing applications," Private Communication.
- [14] Paul Tsuchiya, "Cecilia: A Search Algorithm for Table Entries with Non-contiguous Wildcarding," ftp://thumper.bellcore.com/pub/tsuchiya/cecilia.tar.Z(January 1992).
- [15] Steven McCanne & Van Jacobson, "The BSD Packet Filter: A New Architecture for User-Level Packet Capture," Proceedings of Winter Usenix Conference, San Diego Ca. (January 1993).
- [16] Sally Floyd & Van Jacobson, "Random Early Drop Gateways," IEEE/ACM Transactions on Networking 1,1 (August 1993), 397,413.
- [17] Van Jacobson & Steve McCanne, "TCP-DUMP," ftp://ftp.ee.lbl.gov/tcpdump-*.tar.Z(1990).
- [18] L. Zhang, S. Deering, D. Estrin, S. Shenker & D. Zappala, "RSVP: A New Resource ReSerVation Protocol," *IEEE Network* (September 1993.).

Scaling the Web of Trust: Combining Kerberos and PGP to Provide Large Scale Authentication

Jeffrey I. Schiller Massachusetts Institute of Technology

Derek Atkins Massachusetts Institute of Technology

Abstract

Internet Security has become more important recently as the Internet grows exponentially and security breaches become more publicized. An important area of concern for many Internet users is the privacy and integrity of their electronic files and messages. Phil Zimmermann's Pretty Good Privacy (PGP) provides a general purpose utility for file and message protection. However PGP requires that communicating users be "introduced" to each other. This paper describes a scheme that permits an enterprise using Kerberos to create an automated introducer called the PGP Key Signer Service. Using this service people in the enterprise who have no common acquaintances to act as introducers can be introduced through the Key Signer.

1. Introduction

The Internet Community has grown from a small collection of Computer Science researchers into the International Information Infrastructure. Yet the security technology currently deployed on the Network reflects its original experimental roots. There are many and varied reasons why things are today the way they are. However, one of the key reasons is because most Internet users are unaware of the inherent lack of security on the net. When most people login using their password they assume that only they may use their computer accounts. Similarly when they send electronic mail they assume that only the marked recipients will see the message.

When they receive electronic mail they assume that it originated from the person labeled in the "From" field of the message.

As the Internet has grown, so has the quantity of security problems. Recently these problems have received significant publicity and the public's understanding of the lack of security is growing. With this growth in understanding comes a demand for security.

Several systems now exist to provide security for computers and networks. Two of the more popular security programs are Kerberos [Kerb1, Kerb2] and PGP [PGP]. Kerberos is MIT's real time authentication system, it provides for the security of login sessions and client server transactions. PGP is Phil Zimmermann's "Pretty Good Privacy" public key encryption program. It provides for confidentiality and authentication of electronic mail (e-mail) and other "store and forward" types of transactions as well as encryption of private files.

Each of these systems has its strengths, weaknesses and applicable problem domains. This paper will show how the two can be combined to provide a hybrid security service which neither alone can provide.

In the sections that follow we will provide a brief introduction to PGP and Public Key Cryptography, the underlying technology that makes PGP possible. We will then briefly describe Kerberos. Readers who desire a more thorough background should read the papers referenced above. Finally we will

discuss how PGP can make use of Kerberos in a significant way.

1.1 The Problem with Electronic Mail

SMTP, or Simple Mail Transport Protocol is the Internet standard for the delivery of electronic mail between computer systems. A tried and true protocol, it provides message services to millions of Internet users everyday. Yet it provides for no security services at all. Perhaps more to the point, SMTP servers believe that mail originates from where it claims to.

Forging e-mail messages is child's play. On the MIT campus alone we have seen several independently written shell scripts that permit even the most novice of computer users to send forged messages that are indistinguishable form the real McCoy.

An additional threat to e-mail is that very few networks are protected against eavesdroppers. Recently, bulletins form Carnegie Mellon's Computer Emergency Response Team (CERT)¹ have pointed out many incidents of computer crackers "sniffing" passwords from the Internet [CertSniff]. The same technology that provides for the sniffing of passwords can easily read e-mail in flight as well.

Today the only thing protecting e-mail from these prying eyes is the sheer boredom that most e-mail represents! However as e-mail is used increasingly to discuss confidential corporate plans, commercial transactions and even credit card numbers, it will be a target for network eavesdroppers.

Another important feature lacking in e-mail is the ability to prove after the fact that a message was "signed" by its sender. This ability is needed not only for casual proof of origination, but also for formal uses of e-mail such as electronic purchase orders, bid documents and other commercial interactions. Luckily there is a technology to address this requirement. Digital Signatures permit people to electronically sign documents in a fashion that allows anyone to prove that a document originated from whom it claims to have originated from.

Digital Signatures are provided by Public Key Encryption systems such as the RSA [RSA] system. RSA is used by Internet Privacy Enhanced Mail [PEM] and by PGP. Its use in PGP will be discussed here.

2. PGP

PGP was written by Phil Zimmermann in 1991 in reaction to a move by the U.S. Federal Government to require providers of communication services to provide plaintext copies of encrypted information to the government upon its request. PGP was designed to make this impossible by providing a general purpose cryptographic utility which people can use to protect information that they keep for themselves (say in private files) and share with others (for example electronic mail). As such the primary focus of the development of PGP was on privacy.

PGP provides privacy by using a combination of the IDEA [IDEA] symmetric cipher and the RSA public key system. PGP also includes the ability to make digital signatures. This makes PGP a complete electronic mail security tool. It provides confidentiality, authentication of origination and integrity protection for files and messages.

PGP originally had patent difficulties. Because RSA is patented within the United States, users require a license from Public Key Partners or software from RSA Data Security in order to make use of PGP without the threat of legal action being taken against them. In addition, cryptographic systems, be they hardware or software, currently may not be exported from the United States without a license from the State Department Office of Defense Trade Controls.

Version 2 of PGP was coded in Europe and imported into the United States. Starting in May, 1994 the Massachusetts Institute of Technology (MIT) has been distributing a

¹CERT was founded after the Internet Worm Incident of November 1988. Funded by ARPA, it is operated by Carnegie Mellon's Software Engineering Institute.

version of PGP which includes the RSAREF software toolkit licensed from RSA Data Security. This release of PGP, although not exportable from the United States, provides non-commercial users of PGP a legally usable version. The ViaCrypt company² sells a commercial version that is also licensed from Public Key Partners. The effect is that PGP is just about globally available.

2.1. Public Key Cryptography and its use in PGP

Before we continue we should talk a little about Public Key Cryptography. Invented in 1976 by Whitfield Diffie and Martin Hellman [DiffieHellman], Public Key Encryption is a kind of cipher system where instead of the traditional single encryption key, two keys are employed. One key is used for encryption while the other is used for decryption. Knowing one key does not imply knowledge of the other.

PGP uses the RSA system, named for its three inventors: Ronald Rivest, Adi Shamir and Leonard Adleman. Each user of PGP generates a key pair. This pair consists of a public key and a private key. As their names imply, the public key may be made public while the private key is kept secret by its owner. Both the public and private keys are large values (not simple words or small numbers that can be memorized) so PGP stores them in files. To protect the private key, PGP prompts for a pass phrase during key pair generation. This pass phrase is converted into an IDEA key which is then used to encrypt (using the IDEA cipher) the file that contains the user's RSA private key. A prudent user will protect this file against unwanted perusal even though it is also protected by the pass phrase. Some people go to the length of storing their private key only on a removable floppy which they lock up when not in use.

PGP may be used to encipher a private message between two users if they know each

others' public keys. A sender can send a message to a recipient by encrypting it in the public key of the recipient. The recipient can then decrypt the message by making use of her private key. This is shown graphically in Figure 1. A private key can also be used to construct an unforgeable digital signature which can be affixed to any message. The sender would use her own private key to construct the digital signature, before encrypting the message, and the recipient uses the sender's public key to verify the integrity of a message, after first decrypting it.

² ViaCrypt, 9033 North 24th Avenue, Suite 7, Phoenix, Arizona 85021, USA, Phone: (602) 944-0773, viacrypt@acm.org

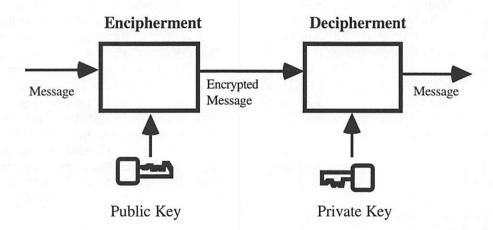


Figure 1: Message Flow and Encipherment

Note carefully that all sensitive operations (reading a private message or creating the unforgeable signature) make use of the private keys. Private keys are never exchanged or disclosed. Public keys are never used for these sensitive operations and therefore it is not a problem for them to be exchanged unencrypted (or published in a public place).

To facilitate the exchange of PGP public keys, a network of **key servers** has arisen that make use of software written by Michael Graff of Iowa State University. The key servers are electronic mail based automated responders. To fetch someone's public key a user sends an electronic message in a particular format requesting the key and the responder automatically sends back the requested key via electronic mail. Brian LaMacchia of MIT has added a World Wide Web server interfaceto this network of servers so keys may be retrieved and stored using popular World Wide Web browsing software.³

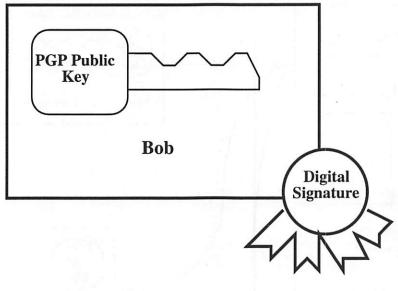
However there is an important issue that needs to be considered. How does someone know that a public key that claims to belong to a particular person in fact does so? After all anyone can generate a key pair and publish the public key along with a claim that the key belongs to, say, the President. There needs to be a way to securely associate a public key with a name.

PGP addresses this problem by the creation of the PGP Web of Trust.⁴ The Web of Trust starts by having two people each generate and exchange public keys and names. Because this is done in person, and presumably between two people who know each other, they can now communicate with each other using signed messages and know beyond a shadow of a doubt that they are talking with each other. The Web grows from this simple one on one association when one of the participants performs a similar key exchange with a third person. An example will illustrate this best.

Assume that Alice and Bob meet in person and exchange public keys. Now Alice and Toni meet later on and exchange keys. Because Alice securely has Bob's key, Alice can provide a "signed" copy of Bob's key to Toni. Alice signs Bob's key by computing a digital signature on it using PGP (which has built in features for doing exactly this exchange). In this example Alice is acting as an Introducer between Bob and Toni. Although they have never met, Alice has introduced them digitally to each other. Figure 2 illustrates this concept.

³http://www-swiss.ai.mit.edu/~bal/pks-toplev.html

⁴Not to be confused with the World Wide Web



Signed by: Alice

Figure 2: Graphical Representation of a Signed Key

PGP permits multiple people to digitally sign an association of a name with a public key. If Bob has another friend, Bill, Bob can ask Bill to sign his key as well. Now if someone gets Bob's key (either from Bob or from a key server) and they trust Bill, they can know that they have Bob's key, even if they never heard of Alice. Figure 3 illustrates this, Bob's key has two signatures on it now, one from Bill and one from Alice.

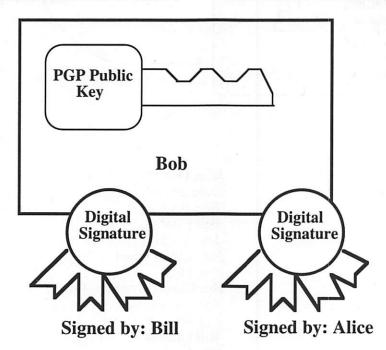
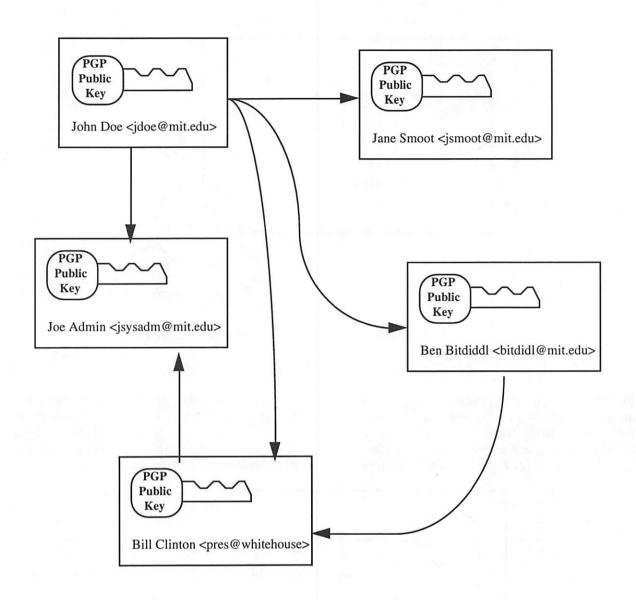


Figure 3: Graphical Representation of a Multiply Signed Key



Jane Smoot Ben Bitdiddl Bill Clinton Joe Admin

John Doe vouches for:

Joe Admin vouches for:

no one

Bill Clinton vouches for:

Joe Admin

Ben Bitdiddl vouches for:

Bill Clinton

Jane Smoot vouches for:

no one

Figure 4: A Sample Web of Trust

As more people sign each others' keys, the Web of Trust builds up. Figure 4 schematically depicts a more complicated situation. The arrows between the keys represent the signatures. So the arrow pointing between John Doe and Jane Smoot represents that John has signed Jane's key.

The fundamental problem with the Web of Trust is that it depends on personal contact between many of the Web participants to build it up. It also depends on all of these participants properly verifying the identity of people whose keys they sign. It is not reasonable to expect a large group of people to be able to perform the necessary verifications without some significant number of them cheating. By cheating we mean that they sign someone's key without really knowing that the key really belongs to them. For example they may sign the key of someone who merely sends it by e-mail. Such a key cannot be trusted because the authenticity of e-mail is suspect in the absence of PGP, which is why we want the Web in the first place!

The Internet's Privacy and Security Research Group was aware of this problem when they designed the Internet standard Privacy Enhanced Mail (PEM). The PEM solution is to create a strict hierarchy of trust. At the root of this hierarchy is the Internet Society which will sign the keys of special organizations called Policy Certification Authorities (PCAs). In turn PCAs will sign the keys of organizations which finally will sign the keys of individuals.

Although this rigid hierarchical approach scales better then the PGP Web of Trust, it requires a fair amount of infrastructure to be in place before two people may use PEM to secure their e-mail. With PGP, if Alice and Bob wish to exchange secure messages and they have no common associates to Web their keys together, they can still meet in person and manually exchange keys. To use PEM, on the other hand, Alice's company will have to be registered with a PCA and be in a position to issue key signatures (called Certificates in the PEM terminology) to Alice. Bob's company will also have to be part of the game. Very few organizations are

currently setup to use PEM. Most likely Alice and Bob are out of luck!

2.2. A Word on Names

PEM requires the use of X.500 style names. These names are of a special form and are not directly printable. By comparison, each PGP name is chosen by an end user. This flexibility is another reason why PGP has found quick acceptance in the community.

Although the PGP program will happily accept any string as the name for a key, Internet users typically use a name that would easily fit in the "To:" field of an e-mail message. For example a PGP key may be named:

John H. Doe <JDoe@xyzzy.com>

where John Doe's e-mail address is JDoe@xyzzy.com. Such a name may be provided to most RFC822 e-mail compliant systems without difficulty.

PGP is oriented toward electronic mail. Electronic mail does not have a strong performance requirement. Basically PGP processes mail messages and needs to be fast enough to run faster then reasonable human perception. However many applications require many transactions per second. Kerberos is designed to work in such an environment.

3. Kerberos

Kerberos was developed by MIT's Project Athena in 1986. It provides for cryptographically based secure real-time authentication of individuals and computers. Kerberos users are given **tickets** when they login which they can then present to network services to prove their identity. Kerberos was designed for real-time client server applications; it is fast and provides protection against replay of sessions and other security threats to a real time system.

In order to make Kerberos fast, and to avoid patent issues during its design, Kerberos today does not make use of Public Key technology. Instead it relies on the U.S. Data Encryption Standard⁵ (DES) [DES] cipher system.

What this means is that with Kerberos one can prove to a particular service that a user is who she claims to be, but one cannot generate a digital document that is digitally signed such that anyone, now and in the future, can verify it.

Kerberos provides authentication services to a large number of users. A Kerberos **Realm**⁶ can support over 100,000 users. At MIT we have approximately 25,000 users in our "ATHENA.MIT.EDU" realm and about 7,000 of them login every day. In general, each organization will have a Kerberos realm of its own. Kerberos can also be used to provide authentication between realms when a shared key is established by the realm administrators.

3.1. A Word on Names

Kerberos names have structure to them. In version 4 (and version 5 as provided by MIT) names have the form name.instance@REALM. Typically a company will use its Internet Domain name

⁵DES is a U.S. Standard Symmetric cipher. Kerberos uses it instead of IDEA simply because IDEA was not around when Kerberos was designed.

as its Kerberos realm name.⁷ The **instance** portion of a name is typically only present in the names of computer services and in special circumstances. Most user names do not have an instance component. Therefore the John Doe of our previous example will likely have a Kerberos name of:

JDoe@XYZZY.COM

4. Combining Kerberos and PGP

Kerberos is designed for an organization size body of users, but doesn't provide digital signatures. PGP on the other hand provides document authentication via digital signatures, but its Web of Trust is cumbersome to use on an organization level. Below we present the design of a service which provides a way to integrate the PGP Web of Trust with the Kerberos authentication model. By doing so an organization can use Kerberos to leverage its use of PGP.

Our design goal is to provide e-mail and document security across an organization the size of MIT⁸ and yet make no modifications to the basic Kerberos or PGP systems.

We do this by introducing a new Kerberos authentication service. We call this service the **PGP Signer** service. The PGP Signer service appears to PGP as just another user who has a public key. To Kerberos it is simply another service which makes use of Kerberos authentication.

To use the PGP Signer service, a user invokes the signer service client program. This program takes the user's public key (and name) and sends this in a Kerberos authenticated transaction to the PGP Signer server. The server, upon receipt of the transaction, compares the name in the PGP public key with the Kerberos authenticated name. This comparison is made using a set of rules that

⁶A realm is the administrative domain of a set of Kerberos servers and the users they authenticate.

⁷Because Kerberos is case sensitive whereas Internet domain names are case insensitive, by convention realm names are always in all upper-case.

⁸The MIT community is roughly 14,000 people, 4,500 undergraduate students, 5,300 graduate students and about 4,000 faculty and staff.

determine if the claimed PGP name is congruent with the authenticated Kerberos name. If they are not congruent the request is rejected; If the names are congruent, then the PGP Signer service signs the PGP public key and name using its private key and returns the result.

To take full advantage of the PGP Signer service, each PGP user in the organization needs to obtain (usually at the same time as they obtain the PGP Signer client program) the PGP Signer's public key and add it to their personal PGP keyring as a trusted signer. When this is done, the PGP Signer can act as a trusted introducer between people. Each user securely meets with the PGP Signer courtesy of Kerberos authentication.

When a group of users wish to exchange email with each other, securing it with PGP, they merely need to use the PGP Signer as their introducer. As a program, the signer never sleeps and is always available to provide service!

4.1. Name Congruency

For PGP names and Kerberos names to be **congruent**, they need to describe the same entity. A simple congruency rule is to require that submitted PGP key names have the structure "Real Name <user@realm>" format. The PGP Signer may choose to ignore the "Real Name" portion of the PGP name and ensure that the portion between the angle brackets is equal to the Kerberos name. So:

John H. Doe <JDoe@xyzzy.com>

would be congruent to:

JDoe@XYZZY.COM

If the PGP signer has access to it a database mapping Kerberos user names to real names, then it can enforce that the supplied real name also corresponds to the actual user name. This permits the PGP Signer to reject names of the form:

Chief Executive Officer and King <jdoe@xyzzy.com>

assuming that John isn't the Chief and King!

4.2. The Security of the PGP Key Signer

The PGP Signer service must run on a secure computer system. This is because it requires access to the PGP Signer's private RSA key. The PGP Signer's Private key is the private key generated along with the PGP Signer's public key, which is published. The PGP Signer never has access to any user's private key. Whoever obtains this private key may construct trusted PGP keys that do not belong to their claimed owner and the organization's security can be compromised.

Organizations that operate Kerberos servers are familiar with the requirement for a physically secure computer, as Kerberos key distribution servers also need to be managed securely. However the PGP Key Signer does not have to be as guarded as the Kerberos server because recovery from the compromise of the PGP Key Signer is not as disastrous.

4.2.1. Operating a Secure Signer Server

Along with using good host security techniques and providing physical security for the PGP Signer, some additional steps taken in advance can make recovery from a compromise easier.

The first step is to keep a copy of the PGP Signer private key in a safe place off-line. A floppy disk can easily store a PGP private key and should suffice for the off-line backup. We recommend that two such floppies be created and stored in safe and secure locations.

The PGP Signer should keep a copy of all public keys that it signs. This database of public keys should be backed up frequently.

⁹The Signer's key will be provided in a file. When PGP processes this file it will automatically recognize that it contains a PGP key and will ask the user if she wishes to add the key to her keyring. PGP will also ask if the key should be "trusted" to sign other keys.

Because it contains no confidential information, this backup of public keys need not be protected against reading, but should be protected against unauthorized modification. An off-line periodic tape backup suffices.

4.2.2. If the PGP Signer is Compromised

Unlike Kerberos, where compromise of the server is a disaster, security can be restored to a compromised PGP Signer with just a little effort.

The first step is to repair whatever host security or physical security problem resulted in the compromise in the first place. Once you are sure that the PGP Signer computing environment is safe, destroy the copy of the public key database and the file containing the PGP Signer private key (these will be "pubring.pgp" and "secring.pgp", the standard PGP keyring files). Restore them from a safe backup. Generate a new PGP Signer key and use this to sign all the keys that are in the backup public key database. Use the old PGP Signer key private key to revoke the old PGP Signer public key. Finally distribute the new PGP Signer public key and ensure that the key revocation certificate¹⁰ that PGP generated for the old key is distributed as well.

Although you will have to "go public" over the compromise of the PGP Signer, the steps that end-users will have to take to recover are minimal.

In the future it may be possible to store the PGP Signer private key in special purpose hardware. BBN manufactures a device that may be used for this purpose. Designed for PEM, the BBN SafekeyperTM may be adaptable for the use with the PGP Signer. The advantage that this hardware brings is that the PGP Signer private key can be stored in it in a fashion that makes it impossible to read it out. An intruder who gains physical access to the PGP Signer server may steal the

Safekeyper, but a "read only" compromise where she secretly learns the PGP Signer private key is not possible.

5. Why not just use Kerberos?

At this point the reader may be wondering why we bother to integrate Kerberos and PGP in the fashion that we do here. Wouldn't it make more sense to simply extend Kerberos so that it can generate digital signatures and provide encrypted e-mail messages?

The answer to this has two parts. The first has to do with creating digital signatures.

To create a digital signature that is verifiable to anyone after it is created requires the use of Public Key Cryptography. Because Kerberos does not make use of Public Key Cryptography it cannot create a digital signature. The best that Kerberos can do is prove to one party that a message is from another party. However, before you can create the equivalent of a digital signature using Kerberos, you need to know the identities of the parties who will be verifying the signature. Using Public Key Technology one can create a signature that anyone, both known and unknown, can verify.

One might consider adding public key cryptographic algorithms to Kerberos in order to implement this function. However one of our design goals was to not make any modifications to the basic Kerberos system. It is also worth noting that if you add public key technology to Kerberos to implement digital signatures for messages you ultimately wind up re-inventing PGP (or PEM).

The other answer is more philosophical and has to do with confidentiality. Because Kerberos does not use Public Key Cryptography, a Kerberos "super-user" can always decrypt a message that was encrypted from one person to another using Kerberos techniques. Perhaps more ominous is that such a "super-user" can be compelled to decrypt messages presented to her under court order.

¹⁰When PGP is instructed to revoke a key it generates a key revocation certificate. It is a datum that instructs any invocation of PGP that receives it to consider the revoked key as invalid.

It is our belief that confidential messages should be exactly that: messages that are only readable by the parties communicating, which usually does not include the government!

6. Project Status

The PGP Signer design is described in this paper. A server is operating at MIT on the Internet host RFA.MIT.EDU. A client program exists for UNIX® systems. As of the time of the writing of this paper (November 1994) MIT has not released the signer system code. However, it is our intention to do so.

7. Future Work

Today PGP has a significant presence in the Internet community, a presence not shared by the standard PEM. However PEM may yet take on an important role as the required certificate infrastructure comes into existence, as it eventually will. A similar **PEM signer** can be created using most of the functions of the PGP Signer, but this signer will issue PEM certificates as well.

8. Conclusion

This paper has described the design of a simple service that permits a digital signature/privacy application, PGP, to leverage and bootstrap its authentication infrastructure from a real-time authentication service, Kerberos. It demonstrates that these technologies are not antagonistic to each other but instead are quite synergistic.

9. References

[IDEA] X. Lai, On the Design and Security of Block Ciphers, ETH Series in Information Processing, v. 1, Konstanz: Hartung-Gorre Verlag, 1992

[CertSniff] Cert Advisory CA-94:01 Ongoing Network Monitoring Attacks, Computer Emergency Response Team, Carnegie Mellon University, Pittsburgth, Pennsylvania (Februrary 3, 1994).

[DES] Federal Information Processing Standards Publication (FIPS PUB) 46-1, <u>Data Encryption Standard</u>, Reaffirmed 1988 January 22 (supersedes FIPS PUB 46, 1977 January 15)

[DiffieHellman] W. Diffie and M. E. Hellman, New Directions in Cryptography, *IEEE Transactions on Information Theory*, v. IT-22, n. 6, November 1976, pp. 644-654

[Kerb1] J G. Steiner, B. C. Neuman, J. I. Schiller, <u>Kerberos: An Authentication Service for Open Network Systems</u>, *Usenix Conference Proceedings* pp191-202, Dallas, Texas (February 1988).

[Kerb2] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer, <u>Project Athena Technical Plan Section E.2.1: Kerberos Authentication and Authorization System, M.I.T. Project Athena, Cambridge, Massachusetts (December 21, 1987).</u>

[PEM] Linn, J., <u>Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures</u>, RFC 1421, Internet Engineering Task Force, (February 1993).

[PGP] P. Zimmermann, <u>PGP User's Guide Volumes I & II</u>, MIT PGP Release 2.6, Cambridge, Massachusetts (May 1994).

[RSA] R. Rivest, A. Shamir, L. Adleman, <u>A</u> method for obtaining digital signatures and public key cryptosystems, CACM, Vol 21 No 2, pp 120-128 (February 1978).

10. Authors

JEFFREY I. SCHILLER received his S.B. in Electrical Engineering (1979) from the Massachusetts Institute of Technology. As MIT Network Manager he has managed the MIT Campus Computer Network since its inception in 1984. Prior to his work in the Network Group he maintained MIT's Multics timesharing system during the timeframe of the ArpaNet TCP/IP conversion. He is an author of MIT's Kerberos Authentication system. Mr. Schiller is the Internet Engineering Steering Group's (IESG) Area Director for Security. He is responsible for overseeing security related Working Groups of the Internet Engineering Task Force (IETF). He is also a member of the Privacy and Security Research Group (PSRG) of the Internet Research Task Force His recent efforts have involved work on the Internet Privacy Enhanced Mail standards (and implementation) as well as releasing a U.S. legal freeware version of the popular PGP encryption program. Mr. Schiller is also a founding member of the Steering Group of the New England Academic and Research Network (NEARnet). NEARnet provides Internet Access to institutions in New England.

DEREK ATKINS is currently a graduate student in Media Arts and Sciences at MIT. He is studying distributed, secure systems for multimedia data distribution. His intrests include, among other things, cryptography, flying, and playing guitar. He can be reached at warlord@MIT.EDU.

Flexible and Safe Resolution of File Conflicts

Puneet Kumar and M. Satyanarayanan Carnegie Mellon University

Abstract

In this paper we describe the support provided by the Coda File System for transparent resolution of conflicts arising from concurrent updates to a file in different network partitions. Such partitions often occur in mobile computing environments. Coda provides a framework for invoking customized pieces of code called application-specific resolvers (ASRs) that encapsulate the knowledge needed for file resolution. If resolution succeeds, the user notices nothing more than a slight performance delay. Only if resolution fails does the user have to resort to manual repair. Our design combines a rule-based approach to ASR selection with transactional encapsulation of ASR execution. This paper shows how such an approach leads to flexible and efficient file resolution without loss of security or robustness.

1 Introduction

Optimistic replication has been shown to be a viable approach to high availability in distributed Unix file systems [15, 3]. It is especially valuable in mobile computing environments, where voluntary and involuntary disconnections are the norm rather than the exception [7]. The Achilles heel of optimistic replication is the need to cope with *conflicts* caused by concurrent updates in different network partitions. Such conflicts can hurt usability if they require frequent manual intervention by users.

In this paper we show how the Coda File System addresses this problem by providing for transparent han-

This research has been supported by the Air Force Material Command (AFMC) and the Advanced Research Projects Agency (ARPA) under Contract F19628-93-C-0193. Support also came from IBM Corporation, Digital Equipment Corporation, and Intel Corporation.

Authors' addresses: School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA, 15213-3891. e-mail: pkumar@cs.cmu.edu, satya@cs.cmu.edu.

dling, or *resolution*, of file conflicts. The essence of our approach is to provide a framework for installing and invoking customized pieces of code called *application-specific resolvers* (ASRs). Each ASR encapsulates knowledge specific to its application. If resolution succeeds, the user notices nothing more than a slight performance delay. Only if resolution fails does the user have to resort to manual repair.

Many practical considerations complicate realization of this simple idea. Users need to be able to control which ASR gets invoked for a specific application. Considerations of security imply the need to restrict the scope of damage caused by an errant ASR. Since intermittent connectivity is common in wireless communication, it is necessary to anticipate failures during the execution of an ASR and to encapsulate its effects in a way that permits easy cleanup. The Coda ASR mechanism addresses these and other related concerns by combining a *rule-based approach* to ASR selection with *transactional encapsulation* of ASR execution. Our implementation confirms that flexible and efficient file resolution is indeed possible without sacrificing security or robustness.

We begin the paper with an overview of Coda and a more detailed rationale for an application-specific approach to file resolution. The bulk of the paper consists of an overview of the ASR mechanism and details on how it achieves flexibility while preserving safety. To illustrate the use of the ASR mechanism, we describe some example ASRs that we have built. We conclude with an evaluation of performance and a discussion of related work.

2 Coda File System

Coda is a descendant of AFS-2 [4] that has *high data* availability as its main goal. Like AFS, it is based on the client-server model, provides a single shared,

location transparent name space, and maintains cache coherence across clients using callbacks. Files are stored in *volumes*, each forming a partial subtree of the name space. Volumes are administrative units, typically created for individual users or projects. At each client, a user-level process called *Venus* manages a file cache on the local disk.

Coda uses two complementary strategies, both based on optimistic replication, to achieve high data availability: server replication and disconnected operation. Server replication allows volumes to be stored at a group of servers called the volume storage group (VSG). At any time, the subset of those servers available is called the accessible volume storage group (AVSG). Disconnected operation arises when the AVSG becomes empty. To prepare for disconnection, users may hoard data in the cache by providing a prioritized list of files. Venus combines explicit hoard information with LRU information to implement a cache management policy that addresses both performance and availability concerns.

Earlier papers [15, 7] provide more details on server replication and disconnected operation. Other papers [8, 10, 11] discuss broader aspects of Coda's approach to conflict resolution, and provide details on mechanisms such as directory resolution that are not covered here.

3 Motivation and Goals

An update conflict arises due to write-sharing of an object from partitioned clients. There is considerable anecdotal evidence and some quantitative evidence [7] to confirm that the average level of write-sharing in personal computing environments is low. This is, of course, what makes optimistic replication viable in such environments.

Unfortunately, certain applications can exhibit much higher than average levels of write-sharing. One example is the use of an online appointment calendar (such as Aldus Datebook [13]) by an executive and her secretary. Another example is the use of an online checkbook (such as Quicken [6]) for a joint account by a couple. Other examples can be drawn from the emerging body of software (such as Lotus Notes or DEC LinkWorks [2]) for computer-supported cooperative work. It is important to note that all of these examples are personal-computing applications; they are not drawn from the online transaction processing domain, where optimistic replication would clearly be inappropriate.

The importance of application assistance in file resolution can be seen by considering the example of a

calendar management program. Suppose an executive and her secretary both make appointments while the former is disconnected. Upon reconnection, Venus detects that the file containing appointments is in conflict. But it has no knowledge of the format of file contents, nor of whether there is really a scheduling conflict. Only code specific to the calendar program can tell, for instance, that appointments for an hour each at 8am and 10am on the same day pose no problem if they are in the executive's office, while those same appointments are impossible to keep if they are in New York and San Francisco.

Even in the absence of computers, conflicts occur in application domains such as the appointment calendar and checkbook examples mentioned above. People are already inured to coping with occasional conflicts in such domains. Hence an acceptable goal for file resolution is to reduce the frequency of manual repair, rather than eliminating it altogether. Total elimination of conflicts is, by definition, an unattainable goal in an optimistically replicated environment.

4 Design Overview

A fundamental design choice pertains to the site of execution of an ASR: should it be executed on a server or on a client? Our choice was to execute ASRs on clients. The primary reason for this decision was to preserve the security model of Coda. Allowing arbitrary ASRs to execute on servers would have violated Coda's basic assumption that servers run only trusted software. Considerations of scalability were a second important factor in our decision. Since the computational resources needed by an ASR may be large, scalability is enhanced by off-loading this burden to clients. A third reason for executing ASRs on clients is the fact that much of the supporting machinery needed to execute an ASR is already present at the client but not at servers. For example, Venus already incorporates the code needed to perform pathname resolution.

Venus performs file resolution *lazily*. An ASR is only invoked in the course of servicing a system call, when Venus discovers that a file has divergent replicas. This is in contrast to an aggressive strategy, whereby execution of ASRs would be performed *en masse* upon recovery from a network partition. Our approach reduces the likelihood of recovery storms, a serious concern in environments with intermittent connectivity. But it does mean that the entire performance cost of ASR execution is incurred by the triggering system call, and can therefore not be hidden from applications and users.

Logically, the Coda ASR mechanism can be viewed as comprising three distinct parts: one part responsible

for *invoking* an ASR when needed, a second part pertaining to *selection* of the correct ASR and a third part responsible for overseeing the *execution* of an ASR. In practice, of course, there is some interdependence between these parts. An underlying consideration in the design of all aspects of the ASR mechanism is the desire to preserve *transparency* from the viewpoint of users.

Viewed from a high level, the ASR mechanism works as follows. On every cache miss, Venus verifies that all replicas of the file being accessed are identical. If Venus detects divergence of replicas it searches for an ASR for this file using rules specified by the user. If an ASR is found, it is executed on the client. The ASR's mutations are performed locally on the client's cache and written back to the server atomically after the ASR completes. The application that requested service for the file is blocked while the ASR is executing. If a failure occurs, ASR execution is aborted and the results of partial execution are flushed from the cache. If no ASR was found or the ASR execution fails, an error code indicating a conflict is returned to the application process.

We elaborate upon this high-level description in the two sections that follow. Section 5 describes those aspects of our design that contribute to flexibility. Section 6 addresses aspects pertinent to safety.

5 Achieving Flexibility

5.1 Invoking an ASR

To execute an ASR on the client, Venus makes a request to a special process called the ASR-starter, as shown in Figure 1. As its name implies, the ASR-starter process is responsible for finding and executing an ASR for a file. An ASR cannot be executed on a client unless it is running an ASR-starter. The functionality of the ASR-starter could have been provided as a routine within Venus. However, we chose not to do so because Venus code is already complex. Implementation and debugging of the ASR-starter was greatly simplified by making it a separate process. The only disadvantage of this approach is slightly higher latency for starting an ASR.

A pool of threads in Venus, called *workers*, are responsible for servicing system calls. Once a worker is assigned to a system call, it is bound to that call until completion. The request to start an ASR is made by a worker when it notices that a file has diverging replicas. This request is made via an RPC, called InvokeASR, to the ASR-starter process. If the request is successful, the process-id (pid) of the ASR is returned to the worker. The worker now blocks, and awaits the

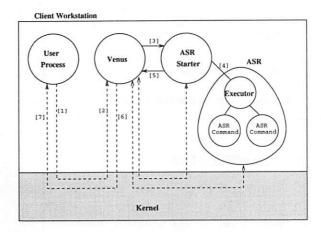


Figure 1: ASR Invocation

This figure shows the message exchange between processes involved in an ASR invocation on a client. Seven events responsible for the ASR invocation and execution are labelled in the Figure: events [1] and [2] are the user request for a file that has diverging replicas; event [3] is the InvokeASR RPC; event [4] is the execution of the ASR commands; the ASR result is returned to Venus via the Result_Of_ASR RPC labelled as event number [5]; events [6] and [7] return the result of the user request. The dotted lines show the message exchange necessary for servicing Coda file system requests made by the ASR-starter, the user process and the ASR processes.

result of the ASR's execution. The user process whose system call is being serviced by the worker thread also remains blocked for this duration. If an ASR cannot be started, resolution fails and a code indicating conflict is returned immediately to the user process.

The InvokeASR RPC has two parameters: the pathname of the file needing resolution, and the identity of the user on whose behalf the ASR is being executed. The pathname is used by the ASR-starter process to find an appropriate ASR. The user identity is used to restrict the privileges of the ASR via the Unix setuid mechanism. The ASR-starter itself runs as root.

Once the ASR completes execution, its result is returned to Venus by the ASR-starter via the Result_Of_ASR RPC. The pid of the recently completed ASR is also returned to Venus so that it can resume the worker waiting for this ASR. If the ASR return code indicates success, the worker retries servicing the user's request. Otherwise it returns an error code, ESYMLINK, to the user application.

Figure 2: Format of rules in a ResolveFile

5.2 Selection of an ASR

Resolution rules are stored in a file named ResolveFile, whose format is similar to a Makefile. The scoping mechanism for ResolveFiles is analogous to lexical scoping in common programming languages. Rules contained in a ResolveFile apply to all files in the subtree rooted at its directory, except where overridden by another ResolveFile lower in the tree. To find the resolution rule for a file, the ASR-starter searches for a ResolveFile upward from the file toward the root. The search stops at the earliest ResolveFile encountered, or at the root.

The advantage of the scoping mechanism is that resolution rules are automatically inherited as new objects are created. For example, a user may have only one set of resolution rules in a ResolveFile in his home directory, which applies to all his files. To simplify sharing of rule files, the ResolveFile entry in a directory can be a symbolic link. For example, an application-writer could provide a ResolveFile for the application's files. All users of this application can share the ResolveFile by creating a symbolic link to it in their personal directory containing the application's data files.

Like Makefiles, ResolveFiles contain multiple resolution rules separated by one or more blank lines. Each rule has the format shown in Figure 2.

The object-list is a non-empty set of object names for which this resolution rule applies. Object-names can contain C-shell wild-cards like "*" and "?", thus allowing one rule to be used for multiple files. For example, a rule that specifies *.dvi in its object-list applies to all files with a .dvi extension.

The dependency-list contains a list of object names whose replicas must be identical before the rule can be applied. This provides a useful check for resolvers that regenerate a file based on the contents of another file. Consider once again the example of a file with a .dvi extension. Its contents can be regenerated by processing the source file with Latex. However, this strategy is usable only if the source file itself does not have divergent replicas, a condition that can be checked au-

tomatically by adding the name of the source file (i.e., the file with the .tex extension) to the dependency-list. In principle it would be possible to recursively resolve objects in the dependency-list. However, to keep the implementation simple, an ASR invocation is aborted if any object in the dependency-list has diverging replicas.

The command-list consists of one command per line. Each command specifies a program to be executed along with its arguments. Since the object-list may contain wild-cards, some file-names that need to be passed as arguments to the resolver programs are not known until the rule is being utilized to invoke the ASR. Therefore, the rule-language provides macros that serve as place holders for these file-names. The language provides three make-like macros, \$*, \$> and \$<. The \$* macro expands to the string that matches the wild-card in the object-list; \$< expands to the pathname of the parent of the file being resolved and \$> expands to the file's name. These macros are expanded dynamically when a rule containing them is used to execute an ASR. Figure 3 shows the macro expansions using a simple example.

.dvi: \$.tex file-resolve \$</\$> latex \$*.tex

For a file /coda/usr/pk/foo.dvi the rule expands to:

file-resolve /coda/usr/pk/foo.dvi
latex foo.tex

And is executed only if the replicas of foo.tex are not diverging.

Figure 3: Macro-expansion of a Resolution Rule

To find a rule that applies to a file foo, the ASR-starter first parses the ResolveFile. The string foo is matched against the names in the object-list of each parsed rule. The first rule containing a match is used as the resolution rule for foo. If no match is found or a syntax error is found in the ResolveFile, the ASR-starter assumes no ASR exists for foo and returns an appropriate error code to Venus.

5.3 Exposing Replicas

Although Venus normally makes replication transparent to user processes, an ASR needs to be able to examine individual replicas of the file being resolved. To allow this, Venus temporarily modifies the name space seen by an ASR. The file being resolved appears to be a directory with each replica appearing as a child of that directory. For example, two replicas of file /coda/usr/pk/foo/would be accessible as /coda/usr/pk/foo/rep1 and /coda/usr/pk/foo/rep2. This in-place explosion of a file into a fake directory only occurs for files being resolved, and only at the client executing the ASR – replication continues to be transparent for all other files and at all other clients.

File replicas in a fake directory are accessible for reading via the normal Unix interface. However, the only mutating operation allowed on the fake directory is the repair pioctl, that takes the name of a replacement file as input. The replacement file can exist in the local Unix file system or it can be one of the replicas of the file being resolved. Its contents are used to atomically set all replicas to a common value. Once this operation succeeds, Venus collapses the fake directory back into a file. The implementation of fake directories makes use of the mount machinery already present in Venus, as elaborated elsewhere [9].

For the duration of the ASR's execution, the volume containing the file being resolved is forced into a special *fakeify* mode. In this mode, any file in that volume with diverging replicas, is converted to a fake directory upon access. This functionality is necessary for an ASR that simultaneously resolves a group of files with diverging replicas. For example, a user's calendar might be maintained in multiple files, and its ASR may need to examine the replicas of all files to perform resolution. Of course, this approach assumes that all files mutated by the application are contained in the same volume.

Since the number and identity of the diverging replicas are not known *a priori*, the rule-language syntax provides three additional macros, [i], [*] and \$#, that serve as *replica specifiers*. [i] is replaced by the name of the i'th replica, [*] by a list of names of all replicas, and \$# by the replication factor of the file. Figure 4 shows the use of these macros with a simple example.

5.4 ASR Execution

The ASR is executed after all macros in the resolution rule have been expanded. Multiple programs may be executed in a single ASR invocation, since a rule's

```
*.cb:
merge-cal-reps $< $# $>[*]

expands to

merge-cal-reps /coda/usr/pk 2 \
cal.cb/server1 cal.cb/server2
```

Figure 4: Macro-expansion of Replica Specifiers

This figure shows the macro-expansion for a file whose replicated pathname is /coda/usr/pk/cal.cb. The file's volume is replicated at two servers, server1 and server2.

command-list can contain more than one command. Since these programs may take an arbitrary long time to execute, they are executed by a separate process, called the EXECUTOR which is forked by the ASR-starter process. The ASR-starter process returns the pid of the EXECUTOR to Venus. Running the ASR via the EXECUTOR frees the ASR-starter process to continue servicing other requests from Venus. Once the ASR completes, its result is returned via the Result_Of_ASR RPC.

The EXECUTOR runs with the identity of the user whose request triggered resolution. It executes the commands in the command-list sequentially, each in a separate process. If any command fails, the EXECUTOR is aborted and an error returned to Venus.

Since Venus runs on multiple hardware platforms, the EXECUTOR must have the ability to choose the binary appropriate for the client machine. To achieve this functionality, it uses the @sys pathname expansion capability of the Coda kernel. The kernel evaluates the @sys component to a unique value on each architecture. This mechanism allows a single resolution command pathname to be used for any client machine. For example, the pathname /usr/coda/resolvers/@sys/bin/merge-cal, translates to /usr/coda/resolvers/pmax_mach/bin/merge-cal on DECstation 5000 clients and to /usr/coda/resolvers/i386_mach/bin/merge-cal on Intel-386 clients.

6 Preserving Safety

6.1 Security

Enforcing security is a critical issue because an ASR is not a piece of trusted system software. A wide range of catastrophes ranging from simple coding errors to full-fledged Trojan horse attacks have to be guarded against. The problem is especially tricky because ASRs are executed transparently. In other words, a user may be completely unaware that an innocent file reference by him caused a malicious ASR to be executed. Coda provides three levels of defense against this problem.

As the first level, which is the default, Coda uses the setuid mechanism to restrict the privileges of an executing ASR. Since the ASR only possesses the privileges of the user who triggered it, damage is limited to those portions of the Coda name space that can be modified by the user.

The next level provides control over which ASRs can be executed by a client. Using the cfs command, a user may specify a list of directories where trusted ASRs can be found. ASR invocation will fail if an attempt is made to execute an ASR from a directory not in the list. Even a trusted ASR is, however, subject to the setuid restrictions mentioned previously. It would be a fairly simple matter to extend this scheme to include a fingerprinting mechanism [18] to detect tampering of ASRs.

The third level prevents ASR execution altogether. As mentioned in Section 5.1, the presence of a ASR-starter process is essential to ASR invocation. If a user configures his client so that the ASR-starter is not run, he is assured that no ASR will ever be executed.

Concerns of security do cause some loss of transparency. Even the least onerous level of security involves some lost opportunities for file resolution because a user needs to have update access, not just read access, on a file to resolve it. Visibility of file conflicts, on the other hand, only requires read access. The second level involves further loss of transparency because it disallows ASRs from untrusted regions of the name space, even though many of them may really be safe. The loss of transparency is, of course, greatest at the third level – in effect, file resolution is avoided altogether.

There are no simple answers to the problem of preserving security while providing transparency in file resolution. Our approach is to allow the user to make the tradeoff, depending on his level of suspicion. Even at the weakest level, however, the user is no worse off than if he were to execute ASR binaries manually.

6.2 Robustness

Misbehaving ASRs can seriously affect the robustness of a client. For example, an ASR whose return code indicates success even though it is unable to resolve the file will cause the Venus worker thread to loop indefinitely: the worker, when resumed, will re-invoke the ASR since the file's replicas are still diverging. Resolvers with programming errors like infinite loops can end up starving user processes of critical resources. Since the number of worker threads is finite, and since a worker is blocked for the duration of an ASR multiple executions of a misbehaving ASR could block all worker threads, resulting in denial of service.

Coda addresses these problems by limiting the execution time of an ASR and the frequency of ASR invocations for an object. In our current implementation, these limits are two minutes and five minutes respectively. The system allows these limits to be changed dynamically for specific objects. Of course, no statically set limit can be perfect: one can always contrive situations where a correct ASR execution is aborted due to one of these limits being exceeded.

6.3 Isolation

To avoid interference between ASRs simultaneously executing on a client, the ASR-starter ensures that at most one ASR executes at a time. A worker thread requesting execution of an ASR is blocked if another ASR is already executing on the client. To avoid deadlock, all locks held by the worker are released before it is blocked. The blocked worker is resumed when the current ASR completes.

Executing only one ASR at a time implies that we cannot currently handle cascaded file resolution across volumes. Consequently, an ASR servicing one volume is aborted if it attempts to access a file with divergent replicas in a different volume. Note that, as described in Section 5.3, an access by the ASR to divergent files within the same volume does not abort the ASR – it merely results in an in-place explosion of those files.

Enforcing serial execution of ASRs at a client only offers local isolation. It does not prevent multiple clients from running an ASR for the same file simultaneously. Coda uses an optimistic concurrency control strategy to handle this problem. Each ASR performs its updates in the client's cache assuming no other client is executing an ASR for the same file. When committing the updates made by an ASR, each server verifies that none of the objects in the ASR's write-set have changed since

the ASR started executing. If this condition is violated, the ASR is aborted. Therefore, if multiple ASRs simultaneously attempt to resolve a file from different clients, only the first to finish will succeed.

To prevent other processes from accessing partial results of an ASR execution, an exclusive lock is held on behalf of the ASR by Venus. This lock applies to the volume containing the file being resolved, and is held for the entire duration of the ASR's execution. Requests from other processes are blocked until the ASR terminates. Venus uses the process group mechanism of Unix to distinguish between ASR and non-ASR requests — each ASR belongs to a new process group, and all processes created by it belong to this group.

6.4 Atomicity

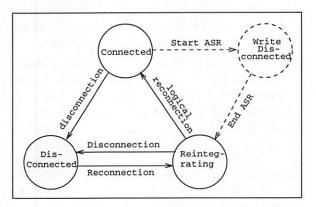
An ASR execution may abort due to a variety of causes such as client or network failure, coding bugs in the ASR, and exceeding ASR time limits imposed by the ASR-starter. Coping with the partial results of an aborted ASR execution can be messy, especially if they lead to further conflicts. To alleviate this problem, Coda ensures that the updates performed by an ASR are made visible atomically. This transactional guarantee can be provided by Venus because it knows the boundaries of an ASR computation (bracketed by InvokeASR and Result_Of_ASR RPC requests) and because it can distinguish requests made by the ASR using the process group mechanism described above.

Venus exploits the mechanism already present for disconnected operation to make ASR execution atomic. To implement disconnected operation, Venus logs all mutations made at a client while it is disconnected from a server. This log is used to *reintegrate* the updates with the server when connection is re-established. The updates are committed at the server atomically, using a local transactional mechanism called *RVM* [16, 17].

To make the updates of an ASR atomic, Venus pretends that the ASR is executing while disconnected. Hence its updates are not written through to the server, but are logged. If the ASR aborts, its updates are undone by purging the log and the modified state in the client's cache. But if the ASR completes successfully, its updates are reintegrated atomically.

The support in Venus for disconnected operation had to be modified to allow servicing of cache misses during ASR execution. Originally, Venus operated in one of three states [7]: connected (also called the hoarding state), disconnected (also called the emulating state), or reintegrating, which is a transient state. We extended Venus to support a fourth state called write-disconnected. This state is a hybrid between the con-

nected and disconnected states. Cache misses are transparently serviced, as in the connected state, but updates are only performed locally and logged, as in the disconnected state. The volume containing the file being resolved is forced into the write-disconnected state before an ASR is invoked by Venus. The modified state transition diagram for a volume is shown in figure 5. Write-disconnected state is similar to the *fetch-only* mode proposed by Huston and Honeyman [5] for disconnected operation in AFS clients.



This figure shows the modified state transition diagram for a volume with the new *write-disconnected* state. The new state and its related transitions are shown in dotted lines/arrows.

Figure 5: Volume State Transitions

Two operations, purge-log and commit-log are provided to purge or commit mutations made by an ASR. Commit-log is invoked if the ASR computation was successful and the server is accessible: the volume transitions into the reintegration state, the ASR's updates are propagated to the server, and then the volume enters the connected state. Purge-log is used to flush the changes made by the ASR. Further details on the implementation of the write-disconnected state are provided in [9].

7 Example ASRs

This section shows the use of the resolution rule language with two example ASRs: a resolver for a calendar management program, and a resolver for a file created by the make utility. The former ASR merges the contents of the diverging replicas. The latter ASR, on the other hand, does not use the contents of the diverging replicas but reproduces the file's data from its source files.

¹The dual of this state is read-disconnected. Both read- and write-disconnected states are collectively referred to as pseudodisconnected states.

7.1 Calendar Application

The cboard application is one of the calendar management programs available in our environment. Using this application, each user can store her appointments in multiple databases. For example, a user may have a personal database for private appointments, and an official database that she shares with her secretary, for recording business appointments. Furthermore, a system database is shared by users to store announcements for public events. The system database and a user's official database, exhibit write-sharing and are thus prone to concurrent partitioned updates.

Each database is maintained in two files, an *events* file and a *key* file. The former, stored in ASCII format, contains one record per event. The key file, on the other hand, is stored in binary format and contains an index of the events file. The index is used for efficient retrieval of a day's events. These two files are distinguished by special name extensions — a .cb extension for the events file and a .key extension for the key file. For example, the files storing the system database, are named system.cb and system.key.

The user interface for the calendar, implemented in Tcl/Tk [12], is shown in Figure 6. A user may browse through a database or mutate it in one of three ways: *insert* a new event, *remove* a cancelled event and *update* a changed event. An event is inserted by appending a new record to the events file. To remove an event, its record is invalidated, i.e. logically removed from the calendar, but not deleted physically from the events file. Finally, an event is updated by invalidating its record and inserting a new one with the updated data. In all three cases, the index is changed and both files are written to disk.

Since each mutation modifies the events and key file, a concurrent partitioned update to the calendar causes both these files to have diverging replicas. Therefore, a resolution rule for the calendar application, as shown in Figure 7, contains both files in its object-list. The dependency-list of the rule is empty, since the ASR does not require any other file to have non-diverging replicas. The ASR is executed in three steps. First, the merge-cal program merges the diverging replicas of the events file and produces a temporary database. The number of replicas and their names, as well as the name of the temporary database are provided as arguments to this program. In the example shown in Figure 7, the temporary database is stored in /tmp/newdb. In the next two steps, the temporary database is used to atomically update the contents of the diverging files using the file-resolve utility. This utility sets the contents of the diverging replicas of a file to a common

value. The new contents are provided in a file whose name is supplied as an argument to this utility. If this argument is missing, the replicas are replaced with an empty file.

The merge-cal program, performs its task as follows. It builds an index of each replica of the events file. The index includes deleted events, even though their records are invalid, so that the resolver can disambiguate between recent deletes and new insertions. The indices are merged using a straightforward algorithm – a unique copy of every valid record is preserved.

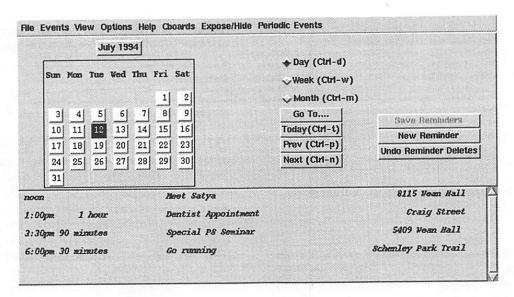
The original calendar application was implemented more than a decade ago. We chose this as one of our test applications due to its wide-spread use in our community. There are approximately 90 regular users of this application. The user interface, when it was originally implemented, was based on terminal input/output. The more recent implementation based on Tcl/Tk and shown in Figure 6 is being used regularly by ten users. We expect this number to increase as the software becomes more stable.

7.2 Make Application: Reproducing a File's Data

Another kind of ASR, commonly used for a file produced by make-like applications, reproduces the file's data from a *source* file. Of course, such ASRs can succeed only if the source file itself is conflict-free.

Common examples of files that could benefit from such ASRs are files with a .dvi or .o extension. The former are created by Latex and the latter are produced by the C compiler. The resolution rules for these two file types are shown in Figure 8. Note that both rules have a non-empty dependency-list, which is typical for ASRs that regenerate the file's contents. The resolver for foo.dvi reproduces its contents by processing foo.tex with Latex, provided foo.tex itself does not have diverging replicas. Similarly, if bar.c has identical replicas, it is processed by the C compiler to generate the contents of bar.o.

Recall, that no mutation except repair is allowed for a file with diverging replicas. Since foo.dvi is updated by Latex while processing foo.tex, its replicas are first truncated using file-resolve. Alternatively, the functionality of the file-resolve utility could be incorporated into Latex. However, our strategy allows us to use the Latex software off the shelf without any modifications. This intermediate step is not necessary for the latter example, since the C compiler allows its output to be redirected to any named file, e.g. /tmp/bar..o in the example above. This



This figure shows the Tcl/Tk based interface for browsing events in the calendar. The menu is used to invoke functions like selecting a database, inserting and deleting events etc. The top frame shows the calendar and highlights those days whose events are being viewed in the lower frame. The user may view events for a day, week or month. The lower frame shows a one line summary for each event. Details for an event can be seen by clicking on its summary line. The application uses a dialog-box to remind users about an event and a window-based form to receive user input.

Figure 6: User Interface for the Calendar Manager

```
*.key, *.cb:

merge-cal-replicas -n $# -f $</$*.cb[*] -db /tmp/newdb

file-repair $*.cb /tmp/newdb.cb

file-repair $*.key /tmp/newdb.key
```

Figure 7: Resolution Rule for the Calendar Application

file is used to set the replicas of bar. o to a common value using file-resolve.

8 Performance

The cost of executing an ASR is visible directly to the user because his request is suspended for the duration of the ASR execution. The time overhead occurs during each of the three parts of the ASR mechanism, i.e. invocation, selection and execution of the ASR. While the time overhead of the first two parts is application-independent, the time overhead of the last part depends on the complexity of the ASR's algorithms and the number of updates made by the application. The purpose of this section is to quantify the overhead of only the first two parts, those concerned with providing flexibility and safety for the ASR mechanism.

To measure this overhead we conducted a series of experiments. An experiment consisted of triggering an ASR and measuring the elapsed time between Venus requesting the ASR and receiving notification from the ASR-starter that the ASR had completed. The ASR was triggered by issuing a stat request for a file with diverging replicas. The ASR consisted of running the Unix command echo without any arguments. Since we are interested in measuring the overhead only for invoking and selecting an ASR, we could have used an empty command-list for the resolution rule. Instead we used a null-program so that our measurements included the cost of a fork system call which would be made even by a minimal ASR.

A measure of the overhead must take into account the fact that the time to find an ASR depends on the lo-

```
*.dvi: *.tex
file-resolve $</$>
latex $*.tex

*.o: *.c
cc -c $*.c -o /tmp/$*..o
file-resolve $> /tmp/$*..o
```

Figure 8: Make Application's Resolution Rules

cation of the ResolveFile with respect to the file being resolved. A longer distance between these files lengthens the time needed by the ASR-starter to find the ASR. To study the effect of changing the depth of the ResolveFile, i.e. the distance between the ResolveFile and the file being resolved, the above experiment was conducted with depth levels from 1 to 12. A depth-level of n implies the ASR-starter had to lookup n ancestral directories to find the ResolveFile.

The experiments were conducted on a DECstation 5000/200 with 32 Megabytes of memory running version 2.6 of the Mach kernel. In each experiment the latency of the ASR execution was measured using a microsecond timer.

8.1 Results

The results of our experiments, shown in Figure 9, confirm that the framework for invoking and selecting an ASR has a small overhead. In most cases, it takes between one-half and one second for the ASR to be invoked and its results returned to Venus.

The minimum overhead of 571.5 milliseconds occurs when the ResolveFile and the file being resolved are in the same directory. Detailed measurements of this experiment configuration show that the RPC request from Venus to the ASR-starter costs 12.5 milliseconds. The fork request that starts the Executor causes a delay of 50 milliseconds. The Executor takes 496 milliseconds to perform its work – 180 milliseconds to parse the ResolveFile and 316 milliseconds to lookup the parent directory for the ResolveFile, lock the volume and fork the echo program. Finally, the RPC from the ASR-starter to Venus costs 12.5 milliseconds.

Although the cost of invoking and selecting an ASR may seem high, it is usually lower than the time needed to execute the resolver. Further, the flexibility of the ASR mechanism combined with the major improvement in usability resulting from ASRs being invoked automatically, rather than manually, renders this cost entirely acceptable.

9 Related Work

The work reported in this paper only represents one part of the overall support for conflict resolution in Coda. Resolution of directories is performed using a servercentric, log-based scheme [10]. The radically different approaches to file and directory resolution arise due to their very different characteristics. Directories are objects whose semantics are entirely known to the system; files, on the other hand, are treated as untyped byte streams. Directories are more critical to availability, because a directory conflict denies access to an entire subtree while a file conflict only denies access to a single object. Finally, to safeguard against structural damage caused by directory corruption, Coda servers never accept entire directory contents from clients as they do file contents; rather, directory updates are performed individually on servers.

These differences lead to Coda's use of very different resolution strategies for directories and files. Directory resolution is performed entirely by servers, although it is triggered by clients. The code for performing directory resolution is part of the trusted system software on a server, in contrast to ASRs which are untrusted. A similarity between directory and file resolution is that both function lazily, and resolve on demand rather than resolving *en masse*. As mentioned earlier, this approach minimizes the likelihood of recovery storms.

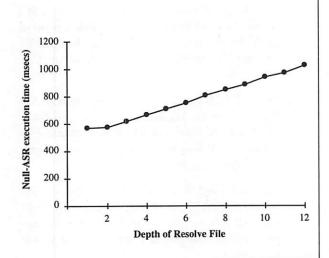
The Ficus File System also uses optimistic replication and provides facilities for automating the resolution of file conflicts [14]. Like Coda, Ficus uses a rule-based approach for selecting a resolver. But it is less flexible along many dimensions. For example, the parameter list to a resolver is assumed to have a fixed format; a Ficus user can have only one personal resolver list; it is not possible to specify that a group of programs are to be executed as one logical resolver, nor can a group of files be resolved together. There are also important differences in the execution models of resolvers in Ficus and Coda. Since Ficus uses a peer-to-peer rather than a client-server model, it is more liberal in its choice of execution site - any site with a replica of a file can run a resolver for it. If one resolver fails, others are tried in succession.

The Ficus design pays less attention to issues of security and robustness. Ficus resolvers are run on behalf of the owner of the file and not the user accessing it. Therefore, a malicious user could cause serious dam-

Resolution rule used for the experiments:

/bin/echo -n

Depth of	ASR Exec.		
ResolveFile	Time (milliseconds)		
1	571.5	(4.9)	
2	576.6	(4.5)	
3	620.8	(5.2)	
4	665.0	(3.2)	
5	708.5	(5.2)	
6	754.1	(13.9)	
7	806.5	(10.9)	
8	847.1	(4.3)	
9	888.9	(4.0)	
10	940.7	(16.7)	
11	970.2	(5.3)	
12	1026.4	(6.8)	



The table shows the elapsed time for executing the null-ASR. The graph shows the increase in overhead with increasing depth of the ResolveFile. Each additional lookup in an ancestral directory takes 44.8 milliseconds. The file being resolved was replicated at two servers. The experiments were performed on a DECstation 5000/200 with 32 Megabytes of memory. The time values in milliseconds are the mean value from nine trials of each experiment. Figures in parentheses are standard deviations.

Figure 9: Execution Time for a Null-ASR

age by using a misbehaved resolver on a file owned by another user. There are no specific mechanisms in Ficus to provide atomicity or isolation. Coda, in contrast, takes these issues much more seriously and provides specific mechanisms to improve safety.

The ASR mechanism in Coda is loosely analogous to a watchdog as proposed by Bershad and Pinkerton [1]: it extends the semantics of the file system for specific files. Of course, since the watchdog mechanism is not specifically intended for conflict resolution, it does not incorporate many important mechanisms needed by us. For example, it does not use a rule-based approach for selection of watchdogs, provide a mechanism for exposing file replicas, or pay particular attention to the issues of security, robustness, isolation and atomicity.

10 Conclusion

The importance of optimistic replication as a technique for providing high availability in distributed systems has been known for over two decades. But the use of this technique in actual systems has been minimal. One reason for this has been the fear of designers that conflicts, an inevitable consequence of optimistic replication, might hurt usability unacceptably. A second reason has been concern that the machinery needed to cope with conflicts might be excessively complex and unwieldy.

Our work puts both these fears to rest in the context of distributed personal computing environments. We have shown how one can build a practical system that provides support for resolving file conflicts. A key aspect of our approach is that the semantic knowledge needed for resolution is provided by application writers rather than being wired into the system. The challenge with this approach is to ensure that security, robustness and other safety properties are not compromised intolerably. This is indeed possible, as we have shown in this paper.

References

- [1] Bershad, B., and Pinkerton, C. Watchdogs Extending the UNIX File System. *Computing Systems 1*, 2 (Spring 1988).
- [2] Eldred, E., and Sylvester, T. A Groupware Duet with Gusto. Client/Server Today 1, 3 (July 1994).
- [3] Guy, R., Heidemann, J., Mak, W., Jr., P., T.W., Popek, G., and Rothmeier, D. Implementation of the Ficus replicated file system. In *USENIX* Summer Conference Proceedings (Anaheim, CA, June 1990).
- [4] Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., and West, M. Scale and Performance in a Distributed File System. ACM Transactions on Computer Systems 6, 1 (February 1988).
- [5] Huston, L., and Honeyman, P. Disconnected Operation for AFS. In Proceedings of the 1993 USENIX Symposium on Mobile and Location-Independent Computing (Cambridge, MA, August 1993).
- [6] INTUIT. User's Guide: Your Day to Day Reference Guide to Quicken, September 1992.
- [7] Kistler, J., and Satyanarayanan, M. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems* 10, 1 (February 1992).
- [8] Kumar, P. Coping with Conflicts in an Optimistically Replicated File System. In Proceedings of the IEEE Workshop on Management of Replicated Data (Houston, TX, November 1990).
- [9] Kumar, P. Mitigating the Effects of Optimistic Replication in a Distributed File System. PhD thesis, School of Computer Science, Carnegie Mellon University, 1994.
- [10] Kumar, P., and Satyanarayanan, M. Log-based Directory Resolution in the Coda File System. In Proceedings of the Second International Conference on Parallel and Distributed Information Systems (San Diego, CA, January 1993).
- [11] Kumar, P., and Satyanarayanan, M. Supporting Application-Specific Resolution in an Optimistically Replicated File System. In *Proceedings of* the 4th IEEE Workshop on Workstation Operating Systems (Napa, CA, October 1993).
- [12] Ousterhout, J. Tcl and the Tk Toolkit. Addison-Wesley, 1994.

- [13] Parkinson, K. Remote Users Get in Sync with Office Files. *Macweek* 8, 28 (July 1994).
- [14] Reiher, P., Heidemann, J., Ratner, D., Skinner, G., and Popek, G. Resolving File Conflicts in the Ficus File System. In *USENIX Summer Conference Proceedings* (Boston, MA, June 1994).
- [15] Satyanarayanan, M., Kistler, J., Kumar, P., Okasaki, M., Siegel, E., and Steere, D. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers* 39, 4 (April 1990).
- [16] Satyanarayanan, M., Mashburn, H., Kumar, P., Steere, D., and Kistler, J. Lightweight Recoverable Virtual Memory. ACM Transactions on Computer Systems 12, 1 (February 1994), 33–57.
- [17] Satyanarayanan, M., Mashburn, H., Kumar, P., Steere, D., and Kistler, J. Corrigendum: Lightweight Recoverable Virtual Memory. ACM Transactions on Computer Systems 12, 2 (May 1994), 165–172.
- [18] Tygar, J., and Yee, B. Strongbox: A System for Self Securing Programs. In CMU Computer Science: 25th Anniversary Commemorative. Addison-Wesley, 1991.

Author Information

Puneet Kumar received a PhD in Computer Science from Carnegie Mellon University in 1994, after a B.S. degree in Computer Science from Cornell University. His PhD thesis was concerned with automating resolution in optimistically replicated distributed file systems. He is one of the original implementors of the Coda File System.

Mahadev Satyanarayanan is an Associate Professor of Computer Science at Carnegie Mellon University. He is currently investigating the connectivity and resource constraints of mobile computing in the context of the Coda File System. Prior to his work on Coda, he was a principal architect and implementor of the Andrew File System. Satyanarayanan received the PhD in Computer Science from Carnegie Mellon University in 1983, after a Bachelor's degree in Electrical Engineering and a Master's degree in Computer Science from the Indian Institute of Technology, Madras. He is a member of the ACM, IEEE, Sigma Xi, and Usenix, and has been a consultant to industry and government.

OBJECTS

Session Chair: Richard Draves, Microsoft Research

NOTES

OODCE: A C++ Framework for the OSF Distributed Computing Environment

John Dilley Hewlett-Packard Laboratories

Abstract

This paper presents a method for developing objectoriented distributed applications using the C++ and DCE technologies. The core of this package is a DCE IDL-to-C++ compiler and a set of C++ classes providing easy access to DCE functionality.

Using this approach we were able to develop more object-oriented distributed applications, and saw a significant decrease in application code size. This contributed to an increase in developer productivity and code maintainability.

1 Introduction

The Open Software Foundation's Distributed Computing Environment (OSF DCE) is an environment for development of distributed applications. DCE consists of a programming environment and a set of services which support distributed computing. The DCE facilities include:

- Communications—The Remote Procedure Call (RPC) mechanism allows processes to communicate across a network using procedure call/return semantics.
 - The RPC component specifies a DCE Interface Definition Language (IDL), a higher-level language used to define the data types and remote procedures associated with an interface. The DCE IDL compiler converts IDL specifications into RPC communication stubs. An application developer must implement the server side routines called by the stubs (the DCE remote procedure "manager functions"). The client stubs provide transparent remote access to the implementation; a call to the server through the client stub looks just like an ordinary procedure call. DCE also defines an exception mechanism, used by RPC to notify client applications of communication or application faults.
- POSIX Threads—A multi-tasking package.
 Pthreads allow a single process to have many threads of control, possibly initiating or serving many RPCs concurrently.

- Naming—The Cell Directory Service (CDS) is a distributed directory service. CDS allows DCE clients to locate servers at run-time in a host-independent manner.
- Security—An implementation of MIT's Kerberos.
 DCE Security provides authentication and authorization for RPC calls so clients and servers can be sure of each other's identity, so data passed via RPC may be encrypted, and so servers can control access to their resources.
- File System—The Distributed File System (DFS)
 provides a shared global file system visible to all
 systems in the cell.
- Time—The Distributed Time Service (DTS) keeps machine clocks synchronized across the network.

DCE services can be accessed programmatically through an application programming interface (API) defined in the DCE AES [1]. They are also accessible through administrative commands, and in some cases through user commands.

1.1 DCE Benefits

Use of the DCE provides application developers with many benefits as compared with network programming using lower-level primitives. Some of the benefits are:

- DCE provides a powerful programming model: application developers can use the familiar procedure call/return model to communicate with remote entities.
- IDL makes network programming simpler: there is no need to write the code that marshals RPC parameters into their network representation. Compiler-generated communication stubs handle all marshaling and unmarshaling for RPC.
- Integration: client applications can use the CDS to provide location-independence; the security service provides secure RPC without requiring developers to be experts in security; integration of threads and exceptions allows easy use of these facilities in RPC applications.

- IDL requires a formal definition of the interface between network objects. The implementation of an object can be in one of a number of languages. Having a formal interface between client and server allows the implementation to be modified independently of the interface; this helps applications to evolve gradually.
- DCE has an implicit object model beyond the pure procedural model exposed by RPC. The object model includes the ability to reference individual objects in the distributed system, and to associate objects with their persistent state, or with a particular implementation in a server.

1.2 DCE Challenges

Along with the benefits, there are a number of challenges posed by DCE: programming complexity often comes with a powerful environment like this. In this case, the DCE services have complex interfaces that a new user needs to learn about before developing DCE applications. For example, there are over 400 DCE interface routines, but only 12 of them are needed in a minimal RPC application. Around 70 DCE routines are used in a more representative application that uses RPC, naming, threads, and security.

Additionally, many of the DCE calls made within an application are similar across applications. This boilerplate code makes applications redundant without adding value, and increases application learning time and long-term maintenance cost.

1.3 Why DCE in C++

The benefits of building systems using object-oriented techniques have been widely discussed in the literature. Korson and McGregor [2] provide a thorough examination of the concepts of object-oriented development. Nicol, et al. [3] explore use of object orientation in distributed systems, and discuss some current distributed object-based systems including the existing DCE object model and OMG's CORBA [4] distributed object system.

We chose to use C++ to encapsulate and abstract DCE functionality for a number of reasons:

- C++ provides object abstraction and data encapsulation, allowing developers to work at a higher level. C++ also supports interface and code reuse.
- C++ is becoming the prevalent language for objectbased systems development. Increasing numbers of C++ class libraries and development tools are becoming available.

- C++ has a clean and natural interface to C, and therefore to the existing DCE implementation.
- The C++ object model is consistent with the object model used in DCE.

These last two factors help to make the resulting system intuitive and therefore easier to learn.

1.4 Project Goals and Requirements

This project set out with three main goals:

- 1. Simplify DCE application development.
- 2. Allow creation of DCE applications in C++.
- 3. Maintain full interoperability with C-based DCE applications.

In order to simplify DCE application development, this project defined a C++ class library that hides DCE's complexity from application developers and a compiler that converts DCE interface definitions into corresponding C++ client and server classes.

The class library provides default behavior for DCE functions such as security and name space registration. This eases application development because suitable behavior is provided automatically by the library. If applications require different behavior, it can be provided within the framework defined by the class library by subclassing and providing a custom implementation. Providing a suitable default implementation reduces the learning time of DCE, and will increase the consistency of distributed applications if they use the same behavior.

Following is a detailed set of requirements for the construction of a class library for DCE.

- Usable—The class library should be easier to learn and use than the underlying technology.
- Standard—The library should build on existing DCE components. It should not require the replacement of any standard DCE component.
- Interoperable—Applications developed using C++ must be able to interoperate with ordinary DCE applications developed in C or other languages supported by DCE.
- Performance—The class library should allow an application to perform within 5% of a C-based DCE application with equivalent features.
- Coverage—While making application development simpler, the library should also provide the ability for the developer to access the full functionality of DCE.

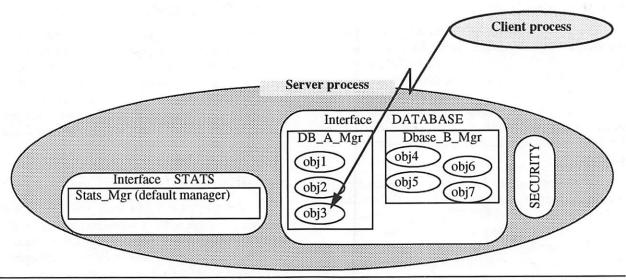


FIGURE 1. DCE Object Model

- Integration—The library should integrate cleanly with the current DCE object model and implementation. It should encapsulate the current DCE behavior in a natural fashion.
- Customizable—The developer should be able to derive from library classes to customize their behavior. This is important for classes that provide policy for interacting with DCE components such as naming and security.
- Extensible—The library should allow developers to derive new classes from those provided to support new capabilities.

2 DCE Object Model

DCE RPC is, obviously, procedure-oriented, but DCE does define a model for object-based application development. The DCE object model is a conceptual one, in which DCE entities are thought of and manipulated as "objects", even though they are not necessarily implemented in an object-oriented language. The DCE C++ class library exposes this model and provides an object-oriented implementation for it.

DCE Definitions

DCE servers provide the services of one or more object implementations to clients through well-defined interfaces. Servers are effectively object managers—the server interacts with the DCE runtime environment to make the services of its objects available to clients.

DCE *interfaces* (written in IDL) define a set of data types and remote operations (procedures). An interface defines how DCE objects can be accessed. The

interface is essentially a contract between the user of an object and the object's implementation; it defines the public signature of the DCE object.

A DCE *object* is a logical entity contained within a DCE server; each DCE server contains one or more DCE objects. Each DCE object *exports* (supports) one or more interfaces. Clients can access DCE objects only through their supported interfaces.

A server can export multiple DCE interfaces. The server in Figure 1 exports three interfaces:

- Its main interface is a database interface. The server contains a set of objects that export this interface. Each object corresponds to a particular entity (tuple or cell) in a database.
- The server also exports a statistical interface to allow clients to retrieve usage statistics about the objects supported by the database interface.
- Finally, the server exports a security interface, used by the database manager functions to control client access to the database based upon client authorization information (implemented in terms of Access Control Lists or ACLs). This interface needs to be exported to allow external programs the ability to view and modify the ACLs.

In addition to supporting multiple interfaces, the server in Figure 1 also has multiple *implementation types* within its database interface. Implementation types A and B each implement the common database interface, but each for a different type of underlying database. The manager functions all have the same signatures and semantics, defined by the IDL, but can have different (private) implementations.

DCE C++ Object Model

We considered two possible approaches to creating a class library for DCE. One approach was to write wrapper classes for each of the basic DCE data types and wrapper functions for each of the interfaces delivered with DCE. These classes and functions would be patterned directly after the data types and functions defined in the DCE specification and would tend to look and operate just like the underlying types and functions.

The other approach, which we chose, was to create a set of classes that build an abstract model of the data and functions needed by a DCE application. These classes were not patterned after data types and procedures, but rather based upon the set of *tasks* or *responsibilities* of the DCE model. The implementations of these classes use the underlying DCE facilities, but do so using an interface independent of those data types; in many cases users need not be aware of details of the underlying data types.

With the wrapper class approach, each underlying interface is typically mapped into a method call, providing little simplification over the current interface. With this approach developers are tied to a particular implementation of the product. Changes in future releases of the base implementation are more likely to require changes to end user code, and possibly to the wrapper classes themselves.

Wrapper classes are still essential in many instances to encapsulate basic data types, even when using a task-based model. One example is the encapsulation of native DCE data types, such as uuid_t, providing the developer a convenient C++ interface to those data types and allowing the use of conversion operators and constructors.

The task-based approach requires more up-front design work by the class library developer, but in the end can yield a system that is more consistent for the application developer, and provides a more natural object model for accessing the product. Task-based systems may also be able to survive changes in their underlying infrastructures, provided the model presented by the new infrastructure remains similar.

We chose to create a set of task-based classes to support the object-oriented DCE (OODCE) model, along with a set of wrapper classes to encapsulate basic DCE data types. The approach is discussed further in the following sections.

3 OODCE Model

The OODCE model preserves the concepts of the DCE object model while providing convenient access to DCE objects. Each DCE object is modeled as one or more C++ objects. Most major user-visible DCE data structures are modeled by C++ classes for convenience of access. With this class library the developer deals primarily with C++ objects in the construction of DCE applications.

3.1 IDL to C++ (OODCE) Compiler

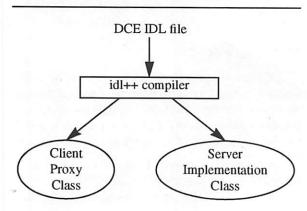
The IDL-to-C++ compiler, idl++, translates a DCE interface definition into interface-specific client and server classes described below. The compiler also generates the regular DCE C header file and client and server stubs used by the RPC mechanism. Since OODCE uses stock DCE client and server stubs, interoperation with DCE applications is assured.

The C++ classes created for OODCE are shown in Figure 2 and described in detail in Section 4.3. Briefly, the client proxy class includes the member functions that allow C++ method calls to invoke the RPCs defined in the IDL file. Application developers must provide the methods of the server implementation class; these methods define the application's behavior, similar to how a C developer implements DCE manager routines.

The data types declared in the header file are left untouched by idl++: no mapping of IDL data types to C++ is necessary since the data types supported by IDL are a subset of the available C++ data types (see the IDL specification [5]). Any data types defined in the IDL file are used by the application developer as C data types.

The methods of client proxy objects make a remote procedure call to the equivalent method in the server

FIGURE 2. IDL++ Generated Classes



implementation object. Proxy objects provide the illusion that the client is calling the remote method directly. In this way, developers can access remote objects using the same programming model they use for making local method calls.

Proxy objects can locate (bind to) implementation objects using the directory name space (CDS), a remote host's endpoint map, an object reference, or via a DCE binding handle. The server's location is determined by how the proxy object is constructed, and via member functions on the proxy object.

3.2 Object References

While it is not currently possible to include C++ objects as arguments in RPCs, it is possible to pass references to C++ (DCE) objects. These DCE object references can allow multiple client proxy objects to refer to the same object instance, and can allow an object to invoke methods on its caller. Object references can also be made persistent and written to stable storage for later reincarnation as proxy objects.

3.3 Object Activation

Instances of DCE server implementation objects can be created at server start-up time to handle client requests. However, if the server will be managing a large number of objects, it is advantageous for the instances to be dynamically activated when requests come in for them. OODCE provides an activation mechanism which allows server developers to define an activation method for their objects.

When a request comes in for an object that is not currently active, this application-defined activation method is called to create and initialize the object. This may involve reading the object's state from persistent storage. A corresponding deactivation method can be defined to allow objects to quiesce themselves after a period of inactivity.

4 The OODCE Class Library

4.1 DCE Framework Classes

DCE framework classes are responsible for providing the DCE object model abstraction. They define a particular object-based view of the distributed system. Client and server implementation classes, generated by idl++, inherit their interface and default behavior from these framework classes. Other framework classes are used to control the behavior of various DCE subsystems, including security, naming, and

threads. Each framework class provides an interface (typically abstract) that defines how that class interacts with the rest of the environment. Many classes also have implementations associated with them to provide the behavior defined by those classes. The framework classes can be specialized by derivation and overriding of virtual functions. Certain classes may not have default implementations; instead they define only the interface (policy) for how to interact with that component of DCE. Instances of those classes must be written by the application developer to provide the behavior for that DCE facility.

The key framework classes are discussed in the following sections. Refer to Figure 3 for a pictorial representation of the classes present in the library.

DCEServer

This class implements the portion of a DCE server that interacts with the DCE environment. A single global instance of the server class manages all the objects and interfaces exported by that server. The DCEServer class is responsible for calling rpc_server_listen to start the server runtime listening for and dispatching incoming RPCs.

Process-global policies, such as for retrieving security keys, can be declared to the server by registering a policy object with the server.

DCEInterfaceMgr

This is the server-side abstract base class. Each interface manager instance is associated with a DCE interface handle (generated by the IDL compiler), a DCE entry point vector (EPV, which contains pointers to the functions that implement the server's manager functions), and optional type and object identifiers for that object instance. The information in the interface manager class is registered with the DCE runtime by the server class.

Derived classes of DCEInterfaceMgr, generated by idl++, add member functions corresponding to the interface-defined remote operations. The derived classes are referred to as server implementation classes. Their member functions are implemented by the server developer.

DCEInterface

This is the client abstract base class. It encapsulates the common client functions for specifying a remote object with which to communicate. The DCEInterface class provides the ability to control [re]binding, security principal information, and authentication policy.

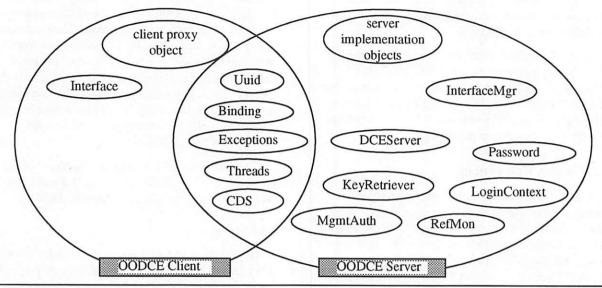


FIGURE 3. Class Library Components

Type conversions are provided from DCEInterface to a DCE client interface handle or to a string. Derived classes of DCEInterface, generated by idl++, provide member functions to access the remote operations. These classes are referred to as client proxy classes.

4.2 Utility Classes

The utility classes encapsulate the basic DCE data types to make their use more convenient in C++, allowing convenient construction and use of these data types. Many classes provide operators to convert to the corresponding DCE C representation (such as to a string or uuid_t), allowing them to be passed directly to DCE calls without need for separate translation. Some examples of the utility classes are DCEUuid, DCEBinding, and DCEPassword.

Another key purpose of these classes is to allow customization of the behavior of the class library. The OODCE framework classes use these utility classes through their defined class interfaces. This allows developers to create derived classes that obey the class interface, yet provide custom behavior.

Security

The class library includes an interface for defining an Access Control List (ACL) manager and ACL storage database. In addition, classes are provided to interface with DCE principals and passwords, to establish and refresh a DCE login context, and to retrieve security credentials from the RPC caller.

Naming

The DCENsi classes model objects in the directory namespace, including CDS names, NSI server entries, and RPC groups and profiles. These classes can be passed to framework classes to identify entries in the name space to use. Applications can also use these classes directly to modify entries in the directory name space.

Threads

The Pthread classes provide C++ access to DCE threads. The classes provide the ability to model pthread mutexes and condition variables, to create and start new threads, to set thread attributes, and to interact with thread specific storage.

4.3 Generated Classes

The OODCE generated classes are created specific for a given interface by the idl++ compiler. This section describes their structure and function in greater detail.

Client Proxy Class

The client-side proxy class is derived from the abstract base class DCEInterface. The proxy class has methods corresponding to the operations (remote procedures) defined in the IDL interface. The client remote operation methods handle the location of the server, and then call the corresponding C stub generated by the DCE IDL compiler. The client methods

also map DCE exceptions returned by the RPC into C++ exceptions.

Server Implementation Abstract Class

For the server, idl++ generates an abstract class derived from DCEInterfaceMgr. This class, called the server implementation abstract class, provides specifications for the member functions corresponding to the remote operations declared in the interface definition file.

Server Implementation Concrete Class

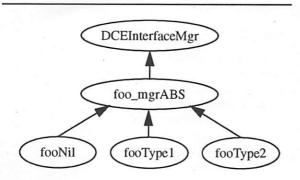
Instantiable (concrete) classes derived from the implementation abstract class must implement all of the member functions defined in the abstract class to provide the operational characteristics of the server. Each of these instantiable classes is of a distinct implementation type (i.e., has an associated manager type UUID). The compiler generates one such class by default with a nil manager type UUID (the nil manager class). If multiple implementation types (and therefore multiple managers) are needed, additional classes can be derived from the abstract manager class. Figure 4 shows the inheritance tree generated by the compilation of an interface foo. The classes fooType1 and fooType2 are developer-defined classes; fooNil is the concrete class generated by the compiler.

Each object (instance) of an implementation class must have its own object UUID and must be registered with the server class so the C++ server stub can locate it when a call comes in for that object.

The implementation classes define manager methods in C++ in the same way the manager functions are implemented in a C-based DCE server. The manager methods use the data passed in as arguments, perform their task, and possibly return data to the caller.

If manager methods throw C++ exceptions they will be caught in the C++ server stub and transmitted

FIGURE 4. Server Class Hierarchy



back to the C++ client (as a DCE-compatible data type) where they are raised again as C++ exceptions for the client to catch.

4.4 Developer-Defined Classes

In addition to all the classes in the class library and those generated by the IDL compiler, application developers have an opportunity to write some classes of their own! Developer-defined classes provide whatever implementation behavior the ultimate application requires. For example, an application might have a graphical user interface (GUI) that collects input and makes RPCs on behalf of the user. In this type of application, the GUI classes would typically control the application—the client main routine would create instances of the desired GUI and DCE client proxy classes, initialize the GUI, register the proxy classes with the GUI, and then await user input. Remote method calls would be made in response to various user inputs. The server implementation classes may also wish to use C++ objects in order to accomplish their tasks.

The OODCE application developer must also implement any of the framework or utility classes for which implementations have not been provided.

5 OODCE Exception Model

By integrating the DCE exception and error models with C++ exceptions we were able to provide a consistent error model and make use of C++ language support for remote and local exceptions [6]. Use of exceptions can reduce application source code size since checking and passing through of error return codes is no longer necessary.

The area of exceptions required quite a bit of work in the design of the class library. One goal of this project was to provide a natural integration between the DCE exception mechanism and C++ exceptions. The C++ exception implementation is far more powerful and better integrated with the language than the DCE exception mechanism. Therefore C++ exceptions were used as the basis for all exception handling in this class library.

The first challenge was that the C++ and C-based exception mechanisms cannot interoperate—an exception thrown or raised by one mechanism cannot be caught and interpreted directly by the other. Furthermore, an exception raised by the C language DCE code cannot be allowed to pass through blocks of C++ code. The DCE exception facility uses setjmp and longjmp; longjmp causes control to pass from one stack

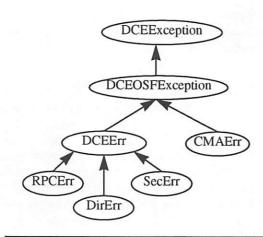


FIGURE 5. Exception Class Hierarchy

frame to another frame, possibly much earlier in the call stack. The trouble is that C++ destructors for automatic (stack) and temporary variables aren't invoked. Allowing a DCE exception to skip over this code can therefore cause consistency problems and memory leaks in the C++ application.

To model DCE exceptions in C++, an abstract base class DCEOSFException was defined; this class specifies the common behavior for all exceptions. Two more base classes, DCEErr and DCECmaErr, were derived from DCEOSFException to model the distinct types of exceptions that can be raised by the DCE and CMA (thread and exception) subsystems. The DCEErr class was further subdivided into RPC, security, and directory service exceptions. Then each specific exception that can be raised by DCE or CMA was modeled as a subclass derived from one of these base classes. Each exception class can be converted into a string (via operator char*()) in order to print a description of the exception.

The choice to model exceptions as individual classes, instead of as a generic class with an exception value, allows individual exceptions to be caught by class—in the C++ exception model any specified data type can be caught. With the generic exception-with-value model, fewer classes would be needed, but if you want to catch a particular exception you would have to catch the generic exception and test its value. This is inconsistent with the more powerful exception mechanism available in C++.

5.1 Server Stubs

Exceptions cannot be directly passed across the network back to the DCE client. Any exceptions raised by the C++ manager methods must be caught and

translated into a data type that can be raised again as a C++ exception in the client. To facilitate this, the idl++ compiler adds a hidden status parameter to the interface definition. That status parameter holds the unique integer value of the exception that was raised. On the client side that integer is thrown again as a C++ exception. Since an integer type is used, only the DCE error_status_t and unsigned32 data types can currently be transmitted back to the client. Any other exception type is mapped to a generic exception code.

5.2 Client Stubs

The C++ proxy method calls the DCE C stub, which can raise a variety of DCE exceptions, including communication status, fault status, and user defined DCE exceptions. To prevent problems caused by the inconsistent DCE and C++ exception models, the DCE C stub call is wrapped by a TRY/CATCH clause. If an exception is caught in the C++ stub, it is rethrown in C++ as the corresponding exception subclass.

6 Application Development

We ported a set of C sample applications from the HP DCE Developer's Toolkit and created a set of new applications in order to experiment with OODCE. The purpose of porting existing applications was to determine the benefits of the OODCE approach. The new applications were necessary to explore the capabilities available in OODCE. Overall we found creation of OODCE applications to be significantly easier than developing ordinary DCE applications.

Our application development environment was HP-UX 8.0 and 9.0, with HP DCE/9000 release 1.1 and 1.2, and HP C++ version 3.20 and 3.40.

6.1 Example code

This section contains sample code from a simple DCE application. This example illustrates what a minimal OODCE application looks like. What follows is fundamentally all the code the application developer needs to write.

IDL file

Client Main

In the above code the constructor for sleeper_1_0 takes a host name and protocol sequence (UDP/IP) to instruct the client how to bind to the server object. The constructor could also have taken a name in the Cell Directory Service (CDS) or an object reference.

Server Main

```
void main()
{
    try {
        // Construct the Sleeper object
        sleeper_1_0_Mgr sleeper;

    // Register with server object
    theServer.RegisterObject(sleeper);

    // activate the server
    theServer.Listen();
} catch (DCEErr& exc) {
        cout << "Caught DCE exception" << (char*)exc;
        exit(1);
}
</pre>
```

The server constructs an instance of a manager (implementation) object and registers it with the global server object. When main exits, the global server object is destructed, at which point it unregisters the interfaces it registered at start-up time.

Manager Code

```
void sleeper_1_0_Mgr::Sleep(long time)
{
    sleep((unsigned int)time);
}
```

The only implementation required is for the single remote operation, Sleep.

6.2 Performance Analysis

One of the requirements for this project was to be within 5% of the performance of an equivalent DCE application. Early experiments have shown a minimal degradation in performance, indicating we are within the 5% figure. The following performance tests looked at the time to construct a binding to the remote server, the time for the first RPC, the average time for ten (10) subsequent RPC calls, and the total time for all these calls. The tables below present the mean time in milliseconds and the 95% confidence interval for the mean. The tests were run on HP 9000 Series 720 workstations configured with 64MB RAM.

In the first test, clients using both DCE and OODCE make a simple RPC request (with a single integer parameter) to an OODCE server.

TABLE 1. RPC to OODCE server

	DCE	OODCE
Binding	42.9 ± 4.2	52.7 ± 2.5
First call	14.2 ± 2.4	3.75 ± 0.24
Other calls	3.19 ± 0.36	3.20 ± 0.45
Total	86.6 ± 4.0	88.6 ± 3.1

The table above shows that the binding time for the OODCE client is longer than for the DCE client. This is due to the type checking being done when constructing a client proxy object: the construction of OODCE objects is type safe, whereas the construction of DCE binding handles is not type safe. Note also that DCE's first call time is much greater than for OODCE. This is because OODCE type checking causes the object to be fully bound to the server, whereas in DCE the resolution of the binding handle happens in the first call.

In the second test, DCE and OODCE clients both contacted a DCE server (in C). This test showed similar performance characteristics as the first test. Additional tests were conducted with client and server on different systems. The results from these tests showed lower call times than the local case, due to the client and server applications being able to run concurrently.

6.3 Results

Porting the existing C applications to C++ took very little time—the main time-consuming tasks were the removal of the code to interface with DCE and the addition of a GUI based on the C++ InterViews user interface class library [7]. Since the interface definitions remained the same, the code dealing with the

data types and remote procedures was highly leveraged. In some of the applications the manager code remained in C, as it was perfectly adequate. This saved development time, and illustrates the integration of the C and C++ environments.

For example, the code size (non-comment source statements) of the OODCE sample applications is about 30% of the corresponding C samples (though the applications are not equivalent: the OODCE sample applications have more capabilities than the corresponding DCE applications).

All the GUI work was done in C++. Since we were using a new UI technology, this was all new code. We found that interfacing the GUI code with the C++ client proxy class was quite natural.

7 Recommendations

We have found it is possible and useful to develop object-oriented distributed applications with C++ and DCE, but there are still some technical issues with doing so. Included below is a set of recommendations intended to smooth the road for others developing object applications using DCE and C++. They are mainly aimed at DCE vendors.

- The sooner thread-safe libraries are available, the
 easier it will be to develop applications using DCE
 threads. In particular, the C++ runtime library, X11
 and GUI technologies built upon it, and commercial products such as databases must be made
 thread-safe. If not, the application must either wrap
 calls to those subsystems or ensure that only a single thread will call them.
- The availability of thread-aware distributed debuggers will also greatly aid in the creation of (working) DCE applications. (This has nothing to do with C++, but will help C++ developers).
- The DCE and C++ exception mechanisms must be reconciled. One alternative is for DCE exceptions to use the same basic mechanism as the C++ exception facility. This would allow consistent exceptions to thrown from one language to the other.
- Users would benefit from a standard framework for developing DCE-based C++ applications. Cooperation between OSF and DCE vendors and standardization on a single approach can help to bring this about. HP intends to work the OSF DCE Special Interest Group (SIG) Object-Orientation Working Group to help make this happen.

8 Related Work

There are many distributed object systems documented in the literature with a variety of goals. The Arjuna system [8][9] focuses on fault tolerance and persistence using a custom RPC mechanism built atop an existing kernel. The Clouds project [10] built a distributed, object-based operating system using a custom microkernel and remote object invocation. Emerald [11] defines a language that uses an object-thread approach to provide distributed object communication. The Spring system [12] provides an object-based distributed system built atop a custom microkernel with a fast, secure object-based IPC mechanism. Schill [13] presents a model for building an object-oriented distributed system within the framework of Open Distributed Processing (ODP) on top of an existing OSIbased infrastructure.

The above are primarily research projects exploring the operation of distributed object systems using custom operating systems or messaging systems. By contrast, our work focuses on integrating C++ with the existing DCE system infrastructure, and on simplifying the use of the existing DCE object model. We do this by providing an extension to the DCE while maintaining compatibility and interoperability with existing C-based DCE systems.

OMG CORBA [4] will eventually address creating distributed object systems in C++; some implementations may run on top of DCE. OMG's IDL provides for interface inheritance, which DCE IDL is currently lacking, and provides a more language-neutral syntax for interface definition. The CORBA runtime environment provides a richer set of object invocation and object passing capabilities than does the DCE environment. However, CORBA today lacks several of the features provided with DCE, such as standard security and naming integrated with the RPC facility, interoperability, and has less distributed computing support in the area of operation semantics and data types supported.

We suggest that our work might assist in the migration of applications from DCE to CORBA by providing an intermediate C++-based distributed object system until CORBA implementations are widely available. Migrating from OODCE to CORBA should be easier than migrating from C or from C++ using explicit DCE calls, because the direct use of DCE is unnecessary. In addition, the use of object-oriented design and programming should assist in making the resulting application more portable into a C++-based CORBA environment than a regular DCE application would be.

There are at least four other current C++-based DCE object systems. One is the DCE++ system proposed by HaL Computers [14]. The HaL solution is fundamentally a wrapper-based approach, providing a C++ interface to the underlying DCE components, but it does not provide a unifying object model. It provides the ability to compile a C++ interface definition into client and server stubs using a custom stub generator; it uses a common base class for all distributed objects.

The second approach is DCE++ from the University of Karlsruhe [15]. This offering provides a new distributed object model based upon fine-grained distributed objects and dynamic object migration. Migration is supported by proxy servers (tombstones), and a per-node location daemon assists in the location of objects (supported by DCE servers). Application classes in this framework are all derived from a common base class; they require a separate, parallel interface and thus an extra compilation. This scheme is somewhat more complex than the one described here, and seems to address a different set of goals.

A third C++ DCE class library proposal has been published by DEC [16]. This DCE RFC describes a mapping from IDL to C++ through extensions to DCE IDL and the attribute configuration file (ACF) facility of the DCE IDL compiler, and therefore modification of the IDL compiler's stub generator. This work approaches the problem by using IDL as the language for describing distributable objects. Among the areas addressed are dynamic object invocation, object migration, and the ability to pass object references between processes. Interface inheritance is also supported at the IDL level. The object migration scheme uses a forwarding mechanism to redirect client requests from a migrated object to its new location. The distinction between local and remote objects is made transparent, allowing a client to be written without needing to know whether an object is a local copy or a surrogate (with a few exceptions). However, this proposal does not address DCE usability by providing a class library framework as ours does.

Finally, Citibank made public a set of classes that make DCE development easier by encapsulating many of the DCE API calls [17]. The Citibank offering defines a new data definition language (YADDL) and a compiler that converts YADDL definitions into C++ classes and DCE IDL definitions. These C++ classes can then be passed in RPCs, like any regular IDL data types can be. In this case the entire object instance is serialized and transmitted; there is no notion of private object state (implementation details, for example) that should not be transmitted.

9 Conclusions

Object-oriented design and development provides a convenient, natural model for distributed systems development—applications can be cleanly modeled as collections objects with remotely callable methods. In specific, the modeling of DCE in C++ has provided significant benefits over the regular DCE interface: the class library and compiler described here significantly reduce the burden of creating DCE applications, and allow creating object-based DCE systems using C++. Most of these benefits could be obtained with an equivalent C library implementation, but the C++ implementation allowed us to expose the DCE object model more naturally. To summarize, the benefits we experienced after use of this technology were:

- Having powerful abstractions on top of DCE allowed us to concentrate on developing the application, not writing code to interface with DCE.
- There is a significant amount of complex code in the library that the user no longer has to write. In particular, the code to deal with the security subsystem is complex and hard to learn. Having easyto-use library calls is a major benefit.
- Application development and debugging time were shortened because basic DCE calls are encapsulated in an already-tested library.
- Having sensible defaults for many DCE values prevented the need for redundant code, allowing code reuse; also having defaults prevents users new to DCE from having to make choices they may not be prepared to make.
- The library provides full coverage of DCE, obviating the need to code to the DCE API. The underlying DCE features are all still accessible by customizing classes provided with the library, so use of the DCE is not limited by this package.
- The C++ DCE application source code size is significantly smaller than the C code. We noted a five-to-one decrease in the actual lines of code that dealt with the DCE environment in the sleeper application. However, the C++ DCE application object size is predictably larger than the C version. The object size grew by 30-40%, mainly due to the addition of the C++ exception mechanism and the OODCE exception classes.
- Having standard policies defined for name space registration and security should assist in making future applications using this library consistent with each other. This will reduce the management

- effort required to maintain a set of client/server applications.
- OODCE is much easier to learn and use than DCE, even for developers who were new to C++. It was easier for them to learn enough C++ to use OODCE than to write a regular DCE application.
- The C++ exception model is more powerful and more useful than the DCE exception model. A greater variety of exceptions can be transmitted more easily across the RPC and handled in the client in a natural, language-supported way.
- C++ is a more powerful language for describing object interfaces and implementations. C++ allows separation of interface from implementation, which is necessary in a distributed environment.

10 Acknowledgments

The idea, initial design, and prototype implementation of this project were mainly due to Jeff Morgan. We also wish to thank Deborah Caswell and Bob Fraley for their advice and contributions to this product, and Amy Arnold, Serena Chang, Jack Danahy, Linda Donovan, Mickey Gittler, John Griffith, Mike Luo, Luis Maldonado, and John Stodieck for their effort in making this system into an HP product. Thanks also go to those who have reviewed earlier drafts of this document, in particular to Rich Friedrich, Joe Martinka, Christopher Mayo, and the USENIX draft reviewers.

11 References

- [1] Open Software Foundation: *OSF DCE Application Environment Specification*. Open Software Foundation, 1992.
- [2] T. Korson, J. McGregor: *Understanding Object-Oriented: A Unifying Paradigm*. CACM Vol. 33, No. 9, September, 1990, p 40-60.
- [3] J. Nicol, C. Wilkes, F. Manola: Object Orientation in Heterogeneous Distributed Computing Systems. IEEE Computer, Vol. 26 No. 6, June 1993, p 57-67.
- [4] Object Management Group: Common Object Request Broker Architecture and Specification. Document Number 91.12.1, Revision 1.1.
- [5] Open Software Foundation: OSF DCE Application Development Guide. Open Software Foundation, 1992.
- [6] M. Ellis, B. Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley. May, 1991.
- [7] M. Linton, J. Vlissides, P. Calder: Composing User Interfaces With InterViews. IEEE Computer, Vol 22, No 2, February 1989.

- [8] S. Shrivastava, G. Dixon, G. Parrington: An Overview of the Arjuna Distributed Programming System. IEEE Software, Vol. 8, No. 1, January, 1991.
- [9] G. Parrington: Reliable Distributed Programming in C++: the Arjuna Approach. USENIX C++ 1990.
- [10] P. Dasgupta, R. LeBlanc Jr., M. Ahamad, U. Ramachandran: *The Clouds Distributed Operating System*. IEEE Computer (to appear).
- [11] R. Raj, E. Tempero, H. Levy, A. Black, N. Hutchinson, E. Jul: *Emerald: A General-Purpose Programming Language*. Software Practical Experiences, Vol. 21 No. 1, January, 1991.
- [12] G. Hamilton, P. Kougiouris: The Spring nucleus: A microkernel for objects. 1993 Summer USENIX, p 147-159.
- [13] A. Schill: OSI, ODP and Distributed Applications: Towards and Integrated Approach. IEEE Global Telecommunications Conference and Exhibition, 1991. p 638-642.
- [14] W. Leddy, A. Khanna: DCE++: A C++ API for DCE. HaL document #030-00209, March 31, 1993, Draft 0.3.
- [15] M. Mock: *DCE++: Distributing C++-Objects* using *OSF DCE*. Proceedings of the International Workshop OSF DCE, Karlsruhe, Germany, October 1993.
- [16] R. Annicchiarico, J. Harrow, R. Viveney: C++ Support in DCE RPC—Functional Overview. OSF DCE RFC 48.1, April, 1994.
- [17] L. Poleshuck: Objtran Programmer's Guide. Citibank Distributed Processing Technology group, December 3, 1993.

Earlier versions of this work were published in the proceedings of the October, 1993 DCE Workshop in Karlsruhe, Germany, and as OSF DCE RFC 49.0.

Biography

John Dilley is a distributed systems architect with Hewlett-Packard Laboratories. His interests include architecture and construction of distributed systems, object location (naming) and distributed directory services, and object-oriented design and development. Mr. Dilley received Bachelor of Science degrees in Mathematics and Computer Science from Purdue University in 1984, and a Master of Science degree in Computer Science from Purdue in 1985. Since then he has been working for Hewlett-Packard in network and systems architecture groups. He was a member of the team that designed the original OODCE prototype.

Mach-US: UNIX On Generic OS Object Servers

J. Mark Stevenson School of Computer Science Carnegie Mellon University Daniel P. Julin
ISIS Distributed Systems

Abstract

This paper examines the Mach-US operating system, its unique architecture, and the lessons demonstrated through its implementation.

Mach-US is an object-oriented multi-server OS which runs on the Mach3.0 kernel. Mach-US has a set of separate servers supplying orthogonal OS services and a library which is loaded into each user process. This library uses the services to generate the semantics of the Mach2.5/4.3BSD application programmers interface (API). This architecture makes Mach-US a flexible research platform and a powerful tool for developing and examining various OS service options.

We will briefly describe Mach-US, the motivations for its design choices, and its demonstrated strengths and weaknesses. We will then discuss the insights that we've acquired in the areas of multi-server architecture, OS remote method invocation, Object Oriented technology for OS implementation, API independent OS services, UNIX API re-implementation, and smart user-space API emulation libraries.

1. Introduction

More and more, operating systems are being developed with a micro-kernel based multi-server structure. Such systems use an operating system kernel to supply system primitives such as tasks, virtual memory, and IPC. Separate servers are used to support derived abstractions (*OS-items*) such as files, TTYs, and pipes. Additional client-side code may be used to do additional OS computation. Examples

of such architectures can be seen in systems such as "V" [Cheriton88], Chorus/MIX [Batlivala⁺92], IBM's "Workplace OS" [Phelan⁺93], OSF1-AD [Zajcew⁺93], FSF "GnuHurd", Sun's "Spring" [Khalidi&Nelson93], and Mach-US from Carnegie Mellon University [Julin⁺91].

The goal of this separation is to achieve several kinds of flexibility in system configuration and development. These include: simplified development and debugging of individual services, replaceable system services, support for multiple API's, efficient use of multi-processor systems, and "loosely coupled" OS configurations. One approach to achieve these goals is to use object-based software with transparent remote method invocation. Mach-US is such a system.

This paper briefly describes Mach-US, the motivations for its design choices and its demonstrated strengths and weaknesses.

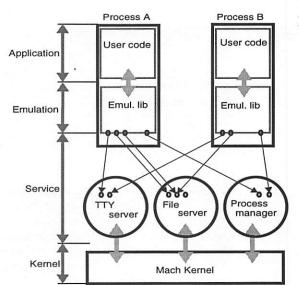
While Mach-US is a natural step in the development of micro-kernel based operating systems, it is revolutionary in many ways. Instead of manipulating previous UNIX implementations or even supporting the basic internal structure or semantics of BSD UNIX, Mach-US is a redesign of the entire OS outside the micro-kernel itself. It supplies a fresh look at a series of OS design options without heavily clouding those options with unnecessary constraints of previous systems. OS design features demonstrated by this system are:

- Multiple System Servers On a Micro-Kernel
- Application Programmer Interface Neutral OS Services
- · Remote Method Invocation for OS Services
- UNIX API Re-Implementation
- Intelligent API Emulation Libraries

This research is sponsored in part by the Advanced Research Projects Agency under contract number DABT63-93-C-0054, and in part by the Open Software Foundation(OSF).

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of ARPA, OSF, or the U.S. government.

Architecture



Revolution, not evolution

Key ideas

- large-granularity servers
- multiple modular servers
- fine-granularity services
- coordination through emul. lib.
- everything out of kernel
- generic services / layering
- client-state processing
- communications through IPC + shared memory

Figure 1: Overall System Architecture

2. Architecture

2.1. Basic Mach-US components

Mach-US is an object-oriented symmetric multi-server OS which runs on the Mach3.0 OS kernel. It is a multi-server system in that it has several servers with orthogonal functions in separate address spaces. These servers supply OS services such as file systems, network support, local sockets and pipes, process management, TTY service, authentication, and configuration.

Mach-US also includes an emulation library which is dynamically loaded into the address space of each user task. This library uses the system servers to generate the semantics of the Mach2.5/4.3BSD application programmers interface (API). It is binary compatible with 4.3BSD. Most servers are essentially independent of the application programmers interface. The few that are UNIX specific were designed to be easily modified or replaced to emulate different API's.

Mach-US is a symmetric OS in that there is no "central" server either to supply an API or for general traffic control. All multi-service actions are controlled by the emulation library.

The overall Mach-US architecture is shown on figure 1.

2.2. System Features

An important aspect of Mach-US is its flexibility. It offers a highly modifiable OS architecture without significant structural impediments to speed. It does this through the following mechanisms:

Object-Oriented Generic OS Interfaces: There are a series of C++ object-oriented interfaces (abstract classes/methods for multiple inheritance) that define semantics supplied by the system servers: access mediation, naming, I/O, network control(OSI-XTI based), and event notification. The various system servers support a combination of these interfaces to do their work and the emulation library uses them to supply a UNIX API.

This uniformity of access makes it easy for new servers to supply additional functionality by sliding into the name-space under the known interfaces. That functionality is then generally available to the system users through their pre-existing software. Additional interface semantics can be achieved through class inheritance and specialization or through the introduction of further orthogonal interfaces.

Modular Services: Different functionality is separated into various servers: configuration, authentication, pathname management, diagnostic, local connections and pipes, UNIX file system, process management, TTYs, and network connections. This separa-

tion makes it simple to develop and debug OS services. One can add and subtract services as needed for a given invocation of the system. Furthermore, a bug in one service doesn't crash or corrupt the entire system.

Object Library/Code Reuse: There is an extensive object library for support/implementation of the various generic interfaces and for use as general server building blocks. This enables faster prototyping of a new server and eases creation of servers from foreign code. Some examples of this support are: namespace manipulation, protection, mapped-files/shared-mem/pager-objects, I/O and bulk-data transfer, client-server binding control and remote method invocation, and extensive interrupt/event/signal support.

3. Interface Model

To capitalize on the idea of using generic services, most high-level functions are defined in terms of an objectoriented framework. Each operating system service is represented by one or more abstract OS-items exporting a well-defined set of operations. Examples of logical OS-items under UNIX are files, pipes, sockets, TTYs, processes, etc. Each such abstraction is represented in Mach-US by a more API neutral OSitem. Each OS-item corresponds to a generic service that is specialized by the emulation library for the more familiar UNIX API semantics. Each server normally implements a large number of similar, but independent, OS-items. Note that the various servers and items are themselves implemented using object-oriented techniques; the word OS-item or item is used to avoid confusion with the actual objects used at the implementation level.

The operations exported by each OS-item are categorized into several independent functional groups which are used throughout the system. Each such group of functions is represented by a specific "standard" interface. Each defines a communication paradigm used for conveying a specific flavor of information under specific semantic and synchronization constraints between the emulation library and the OS-item implementations. The general interface paradigms currently defined are:

Access Mediation: access to all OS-items is mediated through a standard facility using general-purpose access control lists and a uniform representation for user credentials. The requirements of a particular API are handled with special initialization and interpretation of the standard abstractions.

Naming: all OS-items in the system are named and accessible through a uniform name space. There is a common name resolution protocol to navigate the global name space, locate servers and

locate individual items inside a server (e.g., files in a file server, pipes in a pipe server, etc.). It supports user-centered as well as OS-centered naming through the use of prefix tables [Welch86].

I/O: the I/O interface provides a model for the identification and transfer of data. It supports both byte-level and record-level data access as well as sequential and random access operation. Most issues of synchronization are left to the clients (emulation libraries) for maximum flexibility: when necessary, asynchronous style operations are implemented by the client with additional (waiting) threads. Various caching and buffer management policies can be used or created within the same I/O framework.

Network Control: the network control operations deal with the creation and management of transport endpoints, to the exclusion of actual I/O on those endpoints. This interface is largely inspired by XTI[XTI90], with some changes to facilitate sharing of endpoints between multiple clients, and for integration in the uniform name space.

Asynchronous Notifications: this subsystem can be used by servers to deliver notifications, such as UNIX signals, to clients.

Each of these basic interfaces does not directly define the complete functionality exported by any individual OS-item; rather, they correspond to lower-level services that must be combined to define the complete item. For example, network endpoints export operations from both the I/O and network control interface categories.

3.1. Interface Design Guidelines

Interface Objectives: There are many different objectives to be examined when generating new OS interfaces. While these objectives may overlap or partially conflict, we chose to formalize them for Mach-US to help in its development as well as to clarify the system's goals. The previously described interfaces are defined to achieve these goals:

- Separate interface semantics from previous OS API semantics with the intent of supporting multiple different OS APIs, possibly at the same time.
- Make user specific data reside in the emulation library to be passed to servers.
- Ensure that inter-server coordination can be supplied by the emulation library. This avoids forcing unnecessary inter-server communication.
- Enable as much calculation in the emulation library (versus servers) as possible. This lowers the amount of process/server communication and enables more load distribution for multi-processor systems.

- Enable transparent caching of information in userside proxies when useful and safe, again to reduce the amount of process/server communication.
- Supply simple and implementable interfaces that can be used directly by sophisticated users. It is important to make it possible for system server implementors to support these interfaces and to make the power of these interfaces available directly to serious developers for special uses and extra speed.
- Make it fast. This is a partial goal of all the preceding objectives.

Absolute Requirements/Constraints: The above objectives are tempered by these absolute restrictions:

- No data/ports/etc. in the emulation library can be usable to crash a service or other client. If a process directly modifies its emulation library, it may corrupt itself, but it may corrupt no other processes.
- No secure data in the emulation library. A process must not be able to change system-maintained information, that it has no right to change, by modifying its emulation library.
- Must be sufficient to support UNIX API requirements.

4. Client-server model

Mach-US has a remote method invocation package composed of three parts: a run-time system that transports invocations and OS-item references between client and server, a set of client-side objects (*proxies*) that interface between the clients and the run-time system, and a set of server-side objects (*agents*) that interface between the run-time system and the code implementing each OS-item in a server. More details about this package can be found in [Stevenson&Julin94] and [Guedes&Julin91].

4.1. Remote Method Invocation System Properties

The Remote Method Invocation (RMI) system supplies the following features to make OS development possible and simpler:

Communication Flexibility: Through the use of client side proxies [Shapiro86], Mach-US supports various modes of communication within the common invocation facility, so as to use the best approach in each case. There is a default message-passing RMI mechanism and alternate methods such as buffers of shared memory between specific clients and servers. Proxies also support specialized client-side caching and client-side pro-

cessing to maximize the amount of control available to the implementors of OS-items.

Transparency: Code for implementing a feature/method of an OS-item may reside either in the client's address space, a server's address space, or both, and may change on a per item basis. The caller does not need to know the details. All actions against OS-items are standard C++ method invocations.

Forking: The facility provides an efficient and secure mechanism for object reference (binding) inheritance and reconstruction during process creation.

Interruptibility: OS semantics stipulate that some ongoing operations can be asynchronously aborted. The RMI system provides complete support to interrupt pending invocations. Extra care is taken so that methods either complete or are interrupted atomically. A further package is supplied so that servers and the emulation library may control where, when, and how they are interrupted.

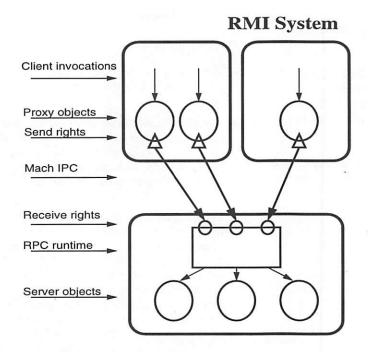
Access mediation and client identification: Each invocation of a specific operation upon an OS-item is subject to access mediation. It is not possible for users to perform unauthorized or forged invocations. Also, the identity of the client performing a given operation is available to the OS-item manager. Although these functions could be provided in an ad-hoc fashion, integrating them with the central invocation facility offers efficiency, simplicity, and uniformity.

Garbage collection: The invocation facility uses Mach3.0 IPC ports [Draves90] with a "no-more-senders" notification to inform a server when there are no more clients of a given OS-item.

4.2. Proxies, Agents, and Agencies

Client view — proxies: Each OS-item is represented on the client side by a special object that functions as a proxy[Shapiro86]. A Mach-US proxy is a body of code that is the client-side part of a given OS-item. All operations on the item are invoked by the client as local operations on the proxy instead of being sent directly to the item's server. In simple cases, the proxy forwards all client invocations to the server via Mach IPC, but in other cases, the proxy may perform most or all of the processing locally, thereby reducing the communications overhead and the load on the server. One typical application is to use a proxy to cache information on the client side of a client-server interface.

A new proxy object is automatically instantiated in a client's address space by the run-time system when an "out arg" object is returned by a remote invocation. The nature and implementation of any given proxy is specified by the designer of each individual server and not by the clients.



- transparent C++ method invocations
- automatic object dispatching
- object references passed as arguments
- proxies can be arbitrarily complex
- garbage-collection through no-moresenders notifications

Figure 2: Remote Method Invocation System

The set of methods exported by a proxy need not be the same as the set of method invocations or messages exchanged between a proxy and its corresponding OSitem manager(server). Only the exported set of methods is visible to clients of the system and constitute the actual OS-item's interface.

Server View — agents and agencies: An agent object is used as a filter between the remote invocation system and the actual server object implementing a given OS-item. Such a server object is called an agency. Remote invocations on each OS-item are dispatched to the user's agent object of the item instead of being dispatched directly to the agency. The agent verifies that a given invocation is allowed under the current access mediation policies and forwards the invocation to the agency for that OS-item. An agent can be queried for further access information as needed.

The structure of the remote invocation system is shown in figure 2.

5. Emulation Model

As mentioned earlier, one of the base goals of the system was to be able to run application binaries without re-compiling or re-linking. This is done by having syscall traps re-directed by the Mach3.0 kernel to the task's emulation library. The emulation library is not a library in the standard sense. It is an executable image that is loaded into the address space of the initial

process at runtime and inherited by its children. Yet, it is not called directly by user processes but instead is invoked by the syscall redirection. While it would be possible for an application to make direct invocations into an emulation library, this has not been done because there has been no real demand for this functionality to date. However, there are special applications that are linked with the system interface libraries used by the emulation library. These applications make calls directly upon the proxy interface.

UNIX API syscalls are implemented in the emulation library itself. Each syscall may make zero, one, or several method invocations against OS-item proxies, and each proxy may or may not need to make a call to a system server to do the requested action. Note that the syscall implementations have the advantage that they actually reside in the address space with the caller, and can access argument memory directly.

The basic responsibilities of the emulation library are:

Syscall: Implements all UNIX API syscalls using OSitem proxies associated with the various system servers.

User State Management: Maintains umasks, sigmasks, seekkeys, and other user-specific parameters.

API Value Translations: Translates values such as *errno* and *uid* from Mach-US general forms into UNIX-specific forms.

- API Semantics Objects: Maintains the API-specific object such as UNIX IO objects, file descriptor table, etc.
- Name Resolution: Follows path names across servers using prefix caching. This caching causes most name lookups to only communicate with the server that supports the OS-item for which one is searching.
- **Forking**: Creates the new child process and reinitializes its state. The process manager is advised of the intent to fork, and of the occurrence of the fork, but is basically an observer.

The gist of forking is that the emulation library creates a new task that shares the state of the current process and then sends it on its way. The problems come with the cloning of that state. In a multi-threaded system, various parts of the emulation library may not be appropriate to just be just copied at any time. These problems are solved through the "cloning" methods supplied by each object in the emulation library.

6. Server Implementation

Any given server supplies a set of OS-items that export some generic OS interfaces to provide their individual functionality. Specialized interfaces may also be exported by an OS-item that has some unique semantics (e.g., process manager). Every server supports a portion of the name space. Its OS-items are found in a tree of directories supplied by the server under the "/servers" directory. A server is responsible for the proxy and agency implementations for each OS-item and generally uses the common agent class to handle OS-item protection.

There is an extensive set of implementation classes used to support the various generic interfaces. Some simple servers (e.g., pipenet and pathname servers) just glue these common implementations together to support their OS-items, whereas other servers (e.g. ufs and net servers) supply their own base item implementations and use the general implementation classes to handle the required client-server communication paradigms. Call-through proxies are automatically generated for each OS-item and some servers inherit these into their more specialized proxies, if any.

The OS servers are written to a UNIX-style environment. While they can never make syscalls, they use a specialized library that provides functionality like malloc, strcpy, and printf. The rest of their work is done by self-contained computation and direct calls to the Mach3.0 kernel. Each server is multi-threaded with individual threads to service individual user requests and to control possible device activity.

6.1. Server Descriptions

Below are descriptions of the specific system servers supplied at this time. There is no reason why other OS-items cannot be supplied by other system servers in the future, and all of the servers supplied need not be used.

- The PathName server manages the root of the name space hierarchy under which all other servers and naming subtrees are mounted. It supplies directory, link, and mount-point OS-items via the naming interface. In particular, it supplies the "/servers" directory.
- The UFS server implements a UNIX file system. It exports the naming interface for directories, mount points, links, etc., and also supports the I/O interface for file objects. Each UFS server currently supports one UNIX disk partition. Several independent UFS servers may be run to support multiple partitions. The emulation library prefix table is used to create the standard UNIX "/" directory tree. Internally, the UFS server contains a C++ vnode style interface [Kleiman86]. It uses the Mach3.0 external pager [Young89] mechanism to map files into a complex proxy to support the read/write semantics for each file.
- The PipeNet server supplies all local IPC endpoints and communication. It is used to implement pipes and UNIX-domain sockets. It supports the naming, block I/O, byte I/O, and network interfaces.
- The Net server supplies network endpoints and communication. It uses a protocol engine from the xKernel project at the University of Arizona [Hutchinson&Peterson91] and supports the same set of interfaces as the PipeNet server.
- The TTY server supplies TTY and PTY OSitems. It supports the naming and I/O interfaces as well as a special interface for bsd_tty_ioctls. Its base implementation is largely derived from 4.3BSD UNIX.
- The Process Manager manages task and taskgroup OS-items. It supports the naming interface and a special task interface. It is responsible for assigning ids, monitoring task lifetime, and forwarding user interrupts. It does not create a client's actual tasks; that responsibility is held by the emulation library.

The following servers do not fit the standard interface model and supply system support functionality. They supply no OS-items to the system namespace.

- The **Configuration** server is used for system startup and general environment support.
- The Authentication server supplies user authentication for OS-item access based on unforgeability

Timing Results					
Test	Mach-UX	Mach-US	Slowdown		
Compile	7.3sec	8.9sec	22%		
Null FTP Get/Put	260Kbytes/sec	240Kbytes/sec	7.7%		
Parallel Compile (1)	39	49sec	26%		
Parallel Compile (10)	146sec	139sec	-4.8%		

Figure 3: Time tests, Mach-UX vs Mach-US

of ports in Mach3.0. This work is derived from [Sansom88].

The Diagnostic server acts as a system output console. Server and emulation library printf, log, and error messages are sent to this server for display on the console.

7. Status and Performance

The Mach-US system is operational and available for distribution to all licensed parties. Near complete distributions are available via anonymous FTP. Mach-US, although not a complete UNIX emulation, is sufficient to support many of the day-to-day tasks performed by typical UNIX users. Most of the common UNIX utilities we know and love (csh, pwd, cc, gnu-emacs, find, ftp(d), telnet(d), inetd, ...) are commonly used. There is full TTY and job control, pipes and signals, file access and access control. One can log into this UNIX system and do useful work today. The entire system has been compiled on itself, using sources stored on a Mach-US file server. Yet, first runs of large applications on Mach-US sometimes uncover problems, generally with API conformance.

The following measurements of Mach-US and its UNIX API are supplied to demonstrate that this system, with its unique and flexible architecture, has acceptable end-to-end performance relative to other UNIXs. They do not offer a detailed timing analysis of the Mach-US system.

The timings given compare the Mach-US UNIX emulation with the same benchmarks for the Mach3.0 single server (Mach-UX) system [Golub+90].

• The compile-test is a Mach classic that compiles nine small programs. One run of this test creates 48 emulated processes which make a total of 9290 system calls. Because of caching and client-side processing in intelligent proxies and the emulation library, there are only 2430 outgoing remote method invocations. Table 3 shows a comparison of compile tests run on an Intel Xpress i486/50Mhz with a 50Mhz memory bus and 24Meg of memory. The Mach3.0 kernel version used was MK83. The Mach-UX times are

for version UX41 and the Mach-US times are for version US50. Our current belief is that the 22% slowdown is caused mostly by new, under optimized Mach-US coding details rather than by its architectural structure. See section 8.1 for further explanation.

• The parallel-compile-test is a related test that runs several compile-tests at the same time. Two tests shown in table 3 were made using one compile test at a time and ten compile tests at a time. These tests were run on Sequent Symmetry hardware using 18 of 20 i386/16Mhz processors and 32Meg of memory. This hardware was used to demonstrate relative strengths of the Mach-US distributed computation on a multi-processor system. Because of recent hardware problems, the timings reported are for slightly older revisions of the system. The Mach3.0 Kernel version was MK78, the Mach-UX version was UX38 and the Mach-US version was US48.

It is significant to note that there is a 31% relative speedup for Mach-US over Mach-UX between the single test and the ten parallel tests. While some of this difference may be caused by arbitrary constraints within Mach-UX, the bulk of the difference can be attributed to a better separation of services in Mach-US than in Mach-UX and a greater client autonomy thanks to the intelligent emulation library and proxy implementations.

• FTP is a very high level test of the Mach-US network service. The FTP client is run on the same system as the compile test. The FTP server (FTPD) is run on a i486/25Mhz with 16Meg of memory using the same kernel and UX as the client system being timed. The FTP server system does not change during the tests. "Get" and "put" operations were done for a 750Kbyte file to /dev/null. /dev/null is used to eliminate disk writes from the tests. The first test was thrown out to ensure that the input file was in the disk cache of each OS. Even though Mach-US uses an xKernel protocol engine, the xKernel uses the same BSD code base used by Mach-UX. The throughput differences demonstrated are fairly small.

The compile-test and other straight-line timings have

shown that Mach-US generally runs 10-25% slower than Mach-UX for high level benchmarks, but shows significant potential for improvement on multiprocessor platforms.

The UNIX API portion of the system is largely un-tuned. This is because the chronically limited Mach-US resources have been concentrated on improving its functionality and robustness. Despite this, some macro-benchmarks show similar or better than those obtained with Mach-UX. There are a number of other known areas where significant performance gains can be achieved through straightforward engineering efforts of bottleneck analysis and RMI information piggy-backing. Additional sophisticated proxies could be implemented to further increase system speed.

The performance gap is real, yet Mach-US achieves the level of performance it does in a more complex, and much richer, environment than other systems available today, and many of the speed differences can be minimized or eliminated using known techniques. Hence, the test results suggest that performance similar to the commonly used Mach-UX single server can be achieved by this multi-server system. We submit that the architecture of Mach-US is not a serious impediment to system usage for speed reasons.

8. Analysis of OS design features

8.1. Multiple System Servers On A Micro-Kernel

The general idea of using multiple separate servers worked very well. The separate servers make debugging new functionality far simpler than dealing with a more monolithic running system. There is little need for concern about corrupting or deadlocking the system under test. One can simply use a standard user-level debugger, such as GDB, on the part of the system in play, leaving the rest of the system running peacefully.

One important research result is that there is almost no need for inter-server communication. The one and only inter-server call in the system is needed to implement keyboard interrupts. This means that a multi-server system need not be crippled by such communication, and does not need to run significantly slower than a single server system. While it could be argued that the server coordination role of the emulation library prevented some inter-server calls, this is not a valid argument. It is true that there are a few functions, such as name resolution, that could be implemented by servers calling servers (for links and mount points). But that approach would yield no fewer inter-address space actions. Mach-US does name resolution by using partial path results for further resolution by the emulation library. There appears to be no real advantage to nested inter-server calls versus iterative

emulation library calls, and iteration gives the emulation library the opportunity of doing additional user space computation and caching.

8.1.1. Mach3.0 Suitability

Running on the Mach3.0 micro kernel itself worked acceptably well. It supplied a rich enough environment that the various system servers were able to do their work and it supported the syscall redirection needed to do the user-space implementation of syscalls. We often run the multi-server beside the Mach-UX single server for even greater ease of debugging.

Mach-US probably exercises the features of the Mach3.0 micro-kernel more than any other system. It became a valuable test case to ensure the correct functioning of the kernel. There were some very winning features of the kernel that helped immensely in Mach-US implementation:

Syscall Redirection is used to cause syscall traps into the kernel to invoke the appropriate code in the emulation library. Without such redirection, re-compilation of all user applications would be required, thus greatly limiting the usefulness of the system.

Threads are used to shepherd user requests through the system servers, syscalls through the emulation library, and wait for various asynchronous events. Since we required more "fairness" than co-routine threads could offer, system supported and scheduled threads were very useful to such a server model. They are essential for optimal use of multi-processor shared-memory systems such as the Sequent system mentioned earlier.

Ports supply two basic features: unforgeable identifiers and IPC endpoints. By combining the concepts, Mach IPC makes it possible for Mach-US to use a port as an OS-item identifier. A separate identifier is used for each user referencing an OS-item. Clients do RMIs to the port itself and the server has a secure ID of the caller. Hence, this basic port model proved to be a powerful weapon in developing an integrated system wide RMI facility.

External Pagers are servers which the Mach3.0 kernel uses to acquire and update pages of data have been logically mapped into a clients address space [Young89]. Hence the external pager mechanism enables sharing of data between the server and a client, where that data is requested by a simple memory reference.

Both the UNIX file server and the process manager act as external pagers. The file proxies pagefault in file pages as needed. The process manager shares memory with each emulation library

to avoid un-necessary server notification of transient process information.

The external pager mechanism supports these two services more efficiently than standard RMI would have.

No-More-Senders port notification can be used to notify a server when there are no remaining external references to a given port. Mach-US uses it for garbage collection of OS-items. It is especially useful since it works automatically for all OS-item references a client holds when the client exits or crashes. This occurres in the same fashion as if the client had explicitly closed the reference.

Scheduling Hand-Off is a Mach kernel mechanism that causes a receiver of a message to continue execution immediately using the remainder of a clients scheduling time slice without waiting to just be rescheduled at some later date. This feature was designed to let small IPCs execute quickly without impacting system fairness, and Mach-US runs significantly faster when such hand-off is utilized. This form of hand-off was a useful step toward the concept of threads that migrate with messages between processes (a feature not in CMU Mach3.0).

There were however some problems when using the Mach3.0 kernel:

Inter Process Communication: The IPC system was not really designed to support a remote method invocation (RMI) paradigm that included interruption. Mach3.0 IPC itself is fairly heavy weight and complex. This comes partly from a "second system effect" and partly from the use of a communication model that attempts to handle every combination of independent message/port transmissions possible, supporting loosely coupled remote procedure calls (RPC) and still making some "fast-path" messages run as fast as possible. Achieving a correct RMI implementation with "fast-path" communication for normal invocations was not simple using the IPC system and there is some reason to believe that a RMI-based IPC could be as effective and easier to use.

The need to interrupt remote method invocations caused headaches. The mechanism for "aborting" long running Mach3.0 requests (e.g., paging and messaging) is crude at best, and deadlocks in some cases. Furthermore, there is no way to identify an individual invocation without taking all invocations off of the fast path or adding inappropriate additional complexity to normal communication. Departure from accepted coding practice was necessary solve these problems. The solutions involved very rare interrupt/restart of uninterrupted invocations, creative use of the "mach_msg" error states, and cautious use of the "abort" mechanism.

Forking: There is no support for forking of kernel entities beyond memory inheritance. There is a real need to support a coherent inheritance model for kernel ports. Furthermore, even memory inheritance can quickly become hard to manage in a complex address space that contains an emulation library, multiple stacks from various threads both in the user application and in that emulation library, out-of-line memory segments created by the memory system, arbitrary buffers, and shared memory areas allocated by various proxy objects or other emulation library components. Long "shadow chains" supporting "copy on write" inherited memory objects became a problem requiring careful memory use.

Even though there were some problems when using the Mach3.0 micro-kernel, it supplied several features that were either useful of essential for the implementation of Mach-US. It was a good and appropriate environment for implementing a multi-server operating system.

8.2. Application Programmer Interface Neutral OS Services

The redesign of the interfaces to OS-items enabled computation to be shifted to the user's address space. This lowered the level of inter-process communication and also speeds computation on multi-processor system configurations.

While we supported the 4.3BSD API on what is believed to be neutral interfaces, that API neutrality is not truly proven. Because of time restrictions, there exists only one emulation library today. However, the same interface calls were often used to support quite disparate syscalls and all API-specific user data is maintained in the emulation library. Hence, there is some evidence that interface neutrality was at least partly achieved.

There are areas where this neutrality was not practical. We did not try to implement a general version of TTY controls. While Mach-US TTYs support the generic ByteIO interfaces, a neutral ioctl system seemed improbable and not worth pursuing. The process manager is also partly API-specific. By the nature of event/signal delivery and its associations with process lifetime, the process manager was implemented to support the specific semantics of 4.3BSD signal/wait. However, its general design and structure are not API-specific, and could be easily modified for another API.

Supporting the generalized interfaces caused additional work, but the redesign effort showed some real fruit. Additionally, to meet the initial design goal of supporting different APIs on a common set of servers, this generality was essential. Such well-defined inter-

faces proved useful when we added new features to the system while creating servers from imported software. Only limited adaptations were needed to fit said interfaces using the support libraries provided. Standard UNIX applications could immediately and transparently use the new features provided.

8.3. Remote Method Invocation For OS Services

The general concept of OS-items appears to be common to many OS definitions. Therefore, it is natural and useful to represent them as programming language objects and to support remote method invocation for their access.

The Mach-US libraries and runtime supply a uniform policy and implementation for transparent remote method invocation, combined client and server side OS-item implementation, interruption, access mediation, forking, and immediate garbage-collection of unreferenced items. Such a powerful unified system for cleanly handling these specific issues for all OS-items was invaluable to the development of a multi-server OS. Using the RMI system was not burdensome to the implementors of OS-items, and it eased their work while supplying the needed flexibility and base implementation to achieve good system speed. The RMI system supplied should be usable and useful to other systems that need to tackle these issues.

8.4. Object Oriented technology for OS implementation

Mach-US uses GNU C++ as its primary implementation language with some plain C where appropriate and some imported C code. Since there has been some controversy on the topics of object oriented (OO) software development and OS implementation, we feel it is appropriate to offer some personal perspective. A large system needs to be carefully architected, independent of whether it is being implemented using OO technology or not. OO technology alone does not solve design problems, yet OO methodology was a very useful tool for interface clarification, data encapsulation, and code re-use through inheritance. Early Mach-US implementation was done in simple C, and we switched to OO technology for those features. Yet it is just a tool, and just like any tool, there are times when it is the right answer, times when it can be made to work, and times where its misuse can be injurious. All of these were experienced during the implementation of Mach-US.

We did not find any real disadvantages to the use of OO methodology for OS development while designing and implementing Mach-US. Its advantages helped in OS development as it would in any other large system, and OO methodology was a good clear choice for representing and implementing OS-items.

8.4.1. Problems with C++

GNU C++ created many problems for us, but was the only appropriate and available alternative. We chose C++ because of its C compatibility, acceptance, availability. We used the "abstract class" feature of C++ to define the generic service interfaces of the system. This made it possible to define clean interfaces for client use, yet leave the implementations to be filled in later by the various implementation classes that lived in their own inheritance trees. Multiple inheritance was used to combine those abstract interface classes and fill in their methods from the implementation classes.

This constituted a logically clean and understandable mechanism that had the flexibility and transparency properties we needed. It relied on the C++ strict type checking to insure interface conformance at every implementation and call site. It was also a bit of a nightmare. These features of C++ are not the most commonly used features. It was sometimes difficult to determine the precise semantics of the combination of multiple inheritance, abstract classes, the "virtual" class property, and object initialization/destruction. More costly problems were that these and other complex features of the language often did not work well, or were only partially functional. The nature of these bugs changed with each revision. We resolved these problems with a combination of syntactic twiddling, work arounds, localized redesign, and replacing parts of the C++ runtime system.

8.5. UNIX API Re-Implementation

A large part of the problem with UNIX API reimplementation is defining what that API is. There are many different documents which describe what syscalls are expected to do, but what is really desired is to have an ever-changing set of applications run on the new system. Each application was written based on what worked for the system upon which it was developed, not the specifications, manual pages, or common knowledge about the system. For example, "Inetd" knows fields of the various socket arguments were not used, and those arguments can't be checked or used by the new system. Our methodology to determine conformance was to run two releases of the common UNIX tools (4.3BSD Tahoe, MtXinu 2.62MSD [MtXinu90]), some formal 4.3BSD compliance tests [Perennial87], and GNU-EMACS. We have built the system from sources upon itself. The largest amount of time spent on any one aspect of the system was spent on API conformance.

While it is difficult, some level of API reimplementation is necessary. OS researchers have a responsibility to explore different OS architectures. In order to be practical, we must write software that supports the semantics of an existing OS, ensuring that we have applications to run. Hence, there is no avoiding partial re-implementation.

We developed our UNIX API emulation library from scratch to fit the initial design objectives of eliminating licensed BSD code and exploring alternate OS-item interfaces that support API neutrality. Conformance was time consuming, yet few of the problems encountered were caused by the Mach-US architecture. The problems encountered would be the same for any system trying to achieve UNIX conformance with a new code implementation.

Our experience implies that an API neutral multiserver symmetric object oriented OS is just as good, or better than other architectures for doing such a UNIX API re-implementation, but such a re-implementation is painful.

8.6. Intelligent API Emulation Libraries:

Except for the difficulty of re-implementing a preexisting ill-defined API, intelligent emulation libraries are quite a powerful tool. Mach-US uses its emulation library to maintain the user-specific API information and implement active client side proxies. For real world test cases, that library makes approximately onefourth as many server calls as a system that makes a server call for every syscall. This difference is important since newer multiprocessor architectures make the relative costs of some inter-processor communications increase. A significant amount of the responsibility for system computation and coordination is shifted to the user process further distributing system computation across the system for such machines.

There are drawbacks:

- It is not always easy to determine if a given bug is in code in the emulation library or in a server. This problem can be overcome with some additional effort and is analogous to the difficulty of determining which "class" in an inheritance tree of any object system may contain a specific bug.
- Forking the emulation library state to a child process is tricky and required a fair amount of overhead and scaffolding to synchronize and reinitialize.
- The emulation library is a part of Mach-US with a complex address space that contains a multitude of threads and stacks, some passed in from the application at syscall time and some that are specific to emulation library execution. This is compounded by the possibility that the application and the emulation library are using different threads packages. Furthermore, it uses all of the interfaces to the other services.

We found that an intelligent of client side library

for system code execution is a complex yet powerful and workable solution to ensure needed flexibility to research the interface between the client process and the system services.

9. Conclusion

We have implemented and demonstrated a system that emulates the UNIX API on top of a micro-kernel with full binary compatibility. Its unique architecture provides many features to enhance the flexibility of system development. This research has provided insights in the following areas:

- Multiple System Servers On a Micro-Kernel is a viable way to strongly partition OS functionality. Such an OS architecture and does not force inter-server calls. Multi-server systems supply flexibility and debugging strengths not available in traditional, more unified OS architectures.
- Application Programmer Interface Neutral OS Services are useful as part of a highly valuable redesign of the interfaces to the OS system services. Yet the current system does not fully prove the API neutrality of its OS service interfaces.
- Remote Method Invocation for OS Services is a simple and consistent point to supply OS developers with the features of: call site transparency, authentication, interruption. It supplies Mach-US with the proxy paradigm for server controlled client-side caching, computation, and communication flexibility. RMI and its associated features has been an essential component for multi-server OS development.
- Object Oriented Technology for OS Implementation is highly useful in the same ways that object-oriented technology is useful for any large software system. Language objects are natural tools to represent OS-items. However, objective technology is only a component of OS implementation and is not a substitute for careful system design.
- UNIX API Re-Implementation is painful and total conformance is difficult to achieve for any system.
 Yet some API re-implementation is required in order to research new OS architectures and still support existing software.
- Intelligent API Emulation Libraries shift work from the OS to the users address space and drastically decrease the need for interaction between the user process and the OS itself.

Mach-US has demonstrated a unique effective system design with many features that enhance system development flexibility and make it a strong platform for further OS research.

10. Further Information

Additional publications and availability information for Mach-US is accessible via the world wide web URL: http://www.cs.cmu.edu:8001/afs/cs/project/mach/public/www/projects/mach_us.html or via anonymousFTP from host mach.cs.cmu.edu in the doc/mach_us and src/mach_us subdirectories.

11. Acknowledgments

Many people at CMU and at the Research Institute of OSF have participated in various stages of the design or implementation of the system. Beside the authors, they are: Robert Baron, Jonathan Chew, Paulo Guedes, Michael Jones, G. Robert Malan, Manish Modh, Paul Neves, Douglas Orr, Richard Rashid, Paul Roy, Richard Sanzi, and Mary Thompson. Further help and support for this paper was supplied by Patricia Jones and Wendy Elcesser.

References

- [Batlivala⁺92] N. Batlivala, et.al. Experience with SVR4 Over CHORUS Usenix Micro-kernels and Other Kernel Architectures Workshop Proceedings, April 1992.
- [Cheriton88] Davic R. Cheriton. The V Distributed System. Communications of the ACM, 31(3):314-333, March. 1988.
- [Draves90] Richard P. Draves. A Revised IPC Interface Usenix Mach Symposium Proceedings, Oct. 1990.
- [Golub+90] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. UNIX as an application program. USENIX Summer 1990 Conference Proceedings, June 1990.
- [Guedes&Julin91] Paulo Guedes and Daniel Julin. Object-Oriented Interfaces in the Mach 3.0 Multi-Server System. Proceedings of the IEEE Second International Workshop on Object Orientation in Operating Systems, October 1991.
- [Hutchinson & Peterson 91] N.C. Hutchinson and L.L. Peterson. The x-Kernel: An architecture for implementing network protocols. In *IEEE Transactions on Software Engineering* 17(1):64-76.Jan.1991.
- [Julin+91] Daniel P. Julin, Jonathan C. Chew, J. Mark Stevenson, Paulo Guedes, Paul Neves, and Paul Roy. Generalized Emulation Services for Mach3.0: Overview, Experiences and Current Status Usenix Mach Symposium Proceedings, November 1991.
- [Khalidi&Nelson93] Y.A. Khalidi and M.N. Nelson. An implementation of Unix on an Object-oriented Operating System. USENIX Winter 1993 Conference Proceedings, January 1993.
- [Kleiman86] Kleiman, S.R. Vnodes: An Architecture for Multiple File System Types in Sun Unix. USENIX Summer 1986 Conference Proceedings, 1986.

- [MtXinu90] MtXinu, Inc. UNIX User's Reverence Manual, 2.6 MSD Version, January 1990.
- [Perennial87] PERENNIAL Inc. PERENNIAL UNIX Validation Suite Manual, April 1987.
- [Phelan⁺93] James M. Phelan, James W. Arendt, and Gary R. Ormsby. An OS/2 Personality on Mach. *Usenix Mach Symposium Proceedings*, April 1993.
- [Sansom88] Robert D. Sansom. Building a Secure Distributed Computer System Thesis CMU-CS-88-141, May 1988.
- [Shapiro86] Marc Shapiro. Structure and encapsulation in distributed computing systems: the Proxy principle. In The 6th International Conference on Distributed Computing Systems, Boston USA, May 1986.
- [Stevenson&Julin94] J. Mark Stevenson, Daniel P. Julin. Client-Server Interaction in Multi-Server Operating Systems: The Mach-US Approach. Carnegie Mellon University Technical Report CMU-CS-94-191, September 1994.
- [Welch86] B. Welch and J. Ousterhout. Prefix tables: A simple mechanism for locating files in a distributed system. In Proceedings of the 6th International Conference on Distributed Computing Systems, pp. 184–189, IEEE, May 1986.
- [Young89] Michael W. Young. Exporting a User Interface to Memory Management from a Communication-Oriented Operation System. Thesis CMU-CS-94-191, September 1994.
- [XTI90] Open Software Foundation. OSF/1 Network Programmer's Guide, 1990.
- [Zajcew+93] R. Zajcew, P.Roy, D. Black, C. Peak, P. Guedes, B. Kemp, J. LoVerso, M. Leibensperger, M. Barnett, F. Rabii, Netterwala. An OSF/1 Unix for Massively Parallel Multicomputers. USENIX Winter 1993 Conference Proceedings, January 1993.

12. Author Information

J. Mark Stevenson received his BS in Math/CS from Carnegie Mellon University in 1983. He was a member of the Osiris multi-processor distributed operating system project at Intel and did distributed engineering drawing image processing with Formtek. During the last five years, he has been on the Mach project at CMU with a primary emphasis on Mach-US. Email: jms@cs.cmu.edu

Daniel Julin has been a member of the Mach system design and development team at Carnegie Mellon University from 1984 to 1993. During that period, he has been primarily involved with the areas of networking, remote inter-process communication, distributed object-oriented systems, security, and operating system emulation. He is now with Isis Distributed Systems, Inc., focusing on high-performance networking, fault-tolerance and consistency in distributed systems. Email: dpj@isis.stratus.com

Events in an RPC Based Distributed System

Jim Waldo, Ann Wollrath, Geoff Wyant, and Samuel C. Kendall Sun Microsystems Laboratories

Abstract

We show how to build a distributed system allowing objects to register interest in and receive notifications of events in other objects. The system is built on top of a pair of interfaces that are interesting only in their extreme simplicity. We then present a simple and efficient implementation of these interfaces.

We then show how more complex functionality can be introduced to the system by adding third-party services. These services can be added without changing the simple interfaces, and without changing the objects in the system that do not need the functionality of those services.

Finally, we note a number of open issues that remain, and attempt to draw some conclusions based on the work.

Introduction

Distributed systems are generally built using one of two distinct communication techniques. The first and most common of these is the distributed analogue of the procedure call, generally called remote procedure call (RPC). The other, less common communication base is the notification from one entity to the other of the occurrence of an event.

Each of these communication techniques has its proponents, and each is appropriate for particular kinds of distributed applications. In this paper, we will discuss how one can build a simple event notification system on top of a remote procedure call system. The distinguishing characteristic of the system described is the simplicity of the underlying protocol for simple event notifications. More complex kinds of event notifications are constructed by the introduction of third-party objects that can be

interposed between supporters of the basic protocol to provide advanced functionality.

RPC based distributed systems

Distributed systems based on Remote Procedure Calls have been around for a considerable period of time [3]. The basic approach is still being used for the construction of distributed systems [15][8][13]. Extensions of the approach introduce remote method invocation on objects [12][5], support for finegrained objects [9], and the automatic location and activation of objects [11].

Throughout these extensions, the mechanisms of the approach have remained essentially unchanged. A call to a remote entity is routed through a surrogate in the address space of the caller. This surrogate is responsible for marshalling the parameters of the procedure (method, function) into a form that can be sent across the wire, and transmitting the result of the marshalling across the wire. On the receiving or server end, a skeleton receives the transmission, unmarshals the result into a form that can be understood by the recipient, and then invokes the local function with the appropriate arguments. Any return values are marshalled by the server, sent across the wire, and unmarshalled by the client surrogate.

This sort of structure lends itself to programming aids. Interface definition languages can be used to define the form of calls from client to server, and compilers for such languages can produce much, or all, of the marshalling and unmarshalling code. The reliance on interfaces to define the communication paths lends this technique to description in terms of objects, and many of the languages used to define these interfaces support a notion of inheritance.

The programming model for an RPC based system is familiar to most programmers. A call to a remote object transfers control to that object, and the calling object blocks until receiving the return of the RPC. With the introduction of threads into the client, only a single thread needs to block. Some systems allow genuine asynchronous calls, but in general the programming model is (by design) as much as possible like that of procedural non-distributed systems.

This leads to a model of the client controlling the interaction, and the server providing some service to that client. Servers in this model are essentially passive entities, waiting for some client to request of them that they do their thing.

The sort of application that fits into this model includes such things as distributed compound documents, in which the various parts of the document are separate objects. The user interface to such a document will call each of the objects, asking it to display itself on the appropriate device when the user moves to that part of the document. The objects themselves react to direct calls from the controlling interface object, which is in turn controlled by some user.

Event based models

A less common model for distributed systems is based on the communication model of event notification. Such systems were pioneered by Isis [2], but other exemplars of this approach to distribution include Teknekron [14], Zephyr [6], and InterStage [7].

The model for such systems is that some significant changes in computational entities making up the system are identified as events. An event might be the change in the price of a stock, or the creation of a new file, or editing the value of a cell in a spreadsheet. Classes of such events are identified as kinds of events that may be significant to others. These other entities can register interest in the occurrence of any event of a particular event kind. If an entity has received such an interest registration, any time the event of that kind occurs it is obliged to send a notification to those entities that have registered interest in that kind of event.

On the surface, such systems allow a much lighter-weight communication mechanism for the distributed system. Notifications need not be synchronous, and indeed most such systems are asynchronous in nature. Complications arise when the ordering of the events from different objects must

be taken into account; considerable work has been done to deal with these orderings [1].

The event paradigm seems somewhat foreign to the object-oriented approach to software. While the computing entities in such a system could be considered objects, the exportation of notifications when the state of one of those objects changes appears to violate the abstraction boundary of the object. The notion of a kind of event appears to show part of the state of the object, which violates the object-based paradigm.

Where the RPC model of distributed computing appears to promote a style of passive objects waiting for some client to request a service, the event notification based systems promotes a style in which objects react to the occurrence of events in their own way. This allows new objects to be introduced to the system that react to events in new ways without changing existing objects.

The style of programming in an event notification based system is similar to that currently popular among developers of user-interface software. Rather than designing the software in a procedural fashion, objects are designed to react to input events from a user.

The sorts of applications that fit well into this model include such things as distributed workflow systems. In such a system, a change in one object (say, the adding of an order to a database) will trigger a notification to a number of other objects (those having to do with inventory maintenance, manufacturing scheduling, or credit checking). Each of the objects that receives the notification will take an action that is appropriate for the object; what action is taken is not known by the object in which the triggering event occurred.

Events in an RPC based distributed object system

Allowing objects to react to changes in the other parts of the system is a useful programming paradigm. To introduce the functionality of an event notification system into a distributed object system based on RPC requires that we take seriously at least the following goals:

 Events must be introduced in such a way that they do not violate the abstraction boundaries of the objects in the system. In particular, the introduction of events should not change the ability of an object to be totally characterized (from the outside) by the set of interfaces that the object supports;

- If possible, introduction of events should not introduce a universal namespace.
 Distributed systems, especially those that are large and developed by multiple programmers in multiple organizations, should not require that all developers adhere to the same naming conventions;
- The basic service should be cheap both in terms of implementation effort and runtime efficiency;
- Complex features should be built on a simple base;
- If different levels of service are used to introduce more complex features, those levels should be completely transparent to those not directly using the service;
- Events should not be introduced as an alternative to RPC, but should only be used to allow functionality that does not fit well into the RPC programming paradigm.

To meet these goals, we have developed a system built around the following basic concepts:

- · Identification of kinds of events
- Registration of interest in a kind of event happening in some object
- Notification of the occurrence of an event. We will look at each of these in turn.

We should note that the only object-specific aspect of what follows is the restriction against violating the encapsulation boundaries of the computational entities involved. The same techniques can be used for distributed systems based on non-object programming approaches.

Event Identification

Our system uses fixed-length identifiers for types or classes of events. These identifiers are obtained from individual objects by calling methods defined for that purpose.

Such methods are part of interfaces defined in the usual way¹. For example, suppose an object supports the exportation of an identifier for an event class that corresponds to changing the name of the object. The method that exports this identifier would most naturally occur in the interface that includes the method called to change the name of the object.

The identifiers used to name an event class are not required to be the same from object to object. All that the system requires is that any particular object issue a single identifier for any particular kind of event and different identifiers for different kinds of events. However, it is possible that the "same" kind of event in different objects will be denoted by different event identifiers, or that identical event identifiers are used by different objects to denote events types that are the "same". We can, however, characterize what it means to say that two event identifiers denote the same kind of event: identifier A and identifier B denote the same kind of event if and only if A and B are returned from calls to the same method (perhaps invoked on different objects).

Making event-kind identifiers relative to the particular object that exports the event class avoids the problem of introducing yet another universal namespace into the infrastructure for distributed systems. Such namespaces can be problematic in a distributed system that has no central authority to ensure that the same name (identifier) is not given to multiple entities. Individual objects can be trusted to make sure that they do not export the same event class identifier for different event classes, and thus act as a local authority on event class identifiers.

Registering interest in an event class

Objects that export interfaces that include methods returning event class identifiers do not, just by that exportation, allow other objects to register interest in those event classes. To do that, the object must export the Event Generator interface.

A short comment on notation. We use the OMG CORBA Interface Definition Language [11] to define our interfaces. This language allows multiple inheritance of interfaces (although we have chosen not to use that feature). We have also adopted the convention of naming the primary interface in any module T (following a convention of Modula-3, our implementation language).

The interface that allows registration of interest in the event classes exported by some object is shown in Figure 1.

Some explanation is in order. The interface refers to three other interface definition files. The interface described in the file "VantageObj.idl" is

In our case, the usual way is to use the OMG CORBA Interface Definition Language (IDL). We will say more about this later.

one supported by all objects in the overall system we are constructing. This interface contains a single method that returns an identifier that (with a high degree of probability) uniquely identifies the object. The interface defined in the file "EventID.idl" describes event class identifiers of the sort discussed in the previous section. The file "EventCatcher.idl" describes the interface that is used to deliver notifications, and will be discussed more fully in the next section.

The interface EventGen::T describes a type of object that is a subtype of the overall VantageObj::T type. The two methods that make up the interface allow objects interested in the occurrence of event classes to register interest in some event class and to cancel such a registration of interest.

```
#include "VantageObj.idl"
#include "EventID.idl"
#include "EventCatcher.idl"
module EventGen{
exception UnknownEvent{};
exception NotRegistered{};
interface T : VantageObj::T {
void
register(
   in EventCatcher::T
           toInform,
   in EventID::T
       eventOfInterest.
   in VantageID::T
       whoIsInterested,
   ) raises (UnknownEvent);
void
unregister (
    in VantageID::T
       whoWasInterested,
    in EventID::T
       eventOfInterest
    )raises
    (UnknownEvent,
   NotRegistered);
   };
```

Figure 1: IDL interface allowing registration of interest in kinds of events.

The register method requires that the caller supply a reference to the object that will receive any notifications of events that are part of this event class, the identifier of the event class of interest, and an identifier for the object that is interested in the event class. On first blush it might appear that identifying either the object that is expressing interest or the object to which the notification is to be sent would suffice. However, the two entities can be distinct, and certainly play logically distinct roles in the protocol. The object expressing interest is the only object that can cancel the registration of interest. However, that object may not be the one that is to receive notifications of occurrences of the event class. Allowing an object to indicate a surrogate for event class delivery allows a third object to enter into the protocol, a feature that we will exploit later to gain more complex functionality.

The unregister method cancels a registration. This informs the object that has exported an event class identifier to stop sending notifications of the event class occurrence to the party indicated in the original registration.

The interface also declares two exceptions that can be raised by methods in the interface. The first of these, UnknownEvent, will be raised by either method when an attempt is made either to register interest in an event class or to cancel registration of interest in an event class using an identifier that is unknown to the object receiving the call.

The second exception, NotRegistered, can be raised by the unregister method if some object attempts to cancel the registration of interest in an event class without having first registered interest in that event class.

It would be possible in all the cases in which an exception is raised to try to "do the right thing", either ignoring the request (in those cases in which UnknownEvent is raised) or by simply returning as if everything is all right (in the case in which NotRegistered is raised). However, it is more likely that the client will know the right thing to do in such exceptional circumstances than that the server will be able to unilaterally determine the correct action, so we chose to return the exception.

Notifications

To receive a notification of the occurrence of an event, an object not only needs to have registered interest in that event class, but must export the interface that allows receipt of notifications. This interface, contained in the file "EventCatcher.idl", is shown in Figure 2.

A notification, according to this interface, is simply a message indicating that some event has occurred. Identification of the event class is made by the combination of the object identifier of the object in which the event occurred and the event class identifier of that event as exported by that object.

The interface also defines a single exception, NoInterest. This will be raised when an object receives notification of an event class which it does not care about. Raising this exception will allow the object that sent the notification to know that it need not send notifications of this event class to that recipient in the future.

A common feature of other notification systems is the ability to return information other than the occurrence of an event as part of the notification. We have not done this, as part of our goal is using notifications only where RPC will not do. If an object that receives a notification wants to know more about the state of the object in which the event occurred, it can obtain that information by making other method calls to that object.

Having all notifications described with a single signature also means that it is simple to insert third parties into the chain of notifications. If different event classes generated different kinds of information in their notifications, it would not be simple to write an agent that accepted notifications and then handed them off to others.

```
#include "VantageObj.idl"
#include "EventID.idl"

module EventCatcher{
exception NoInterest{};
interface T : VantageObj::T {
  void
  notify(
      in VantageID::T from,
      in EventID::T whatEvent
      ) raises (NoInterest);
    };
};
```

Figure 2: IDL interface for objects that can receive notifications of event occurrences

Implementing the Simple Interfaces

While we use the OMG CORBA IDL to describe the interfaces in our system, we do not use a CORBA-based distributed object management facility (DOMF) in our implementation. Our implementation language is Modula-3 [10], and our distribution substrate is the Network Objects package distributed with the DEC SRC version of that language [4], enhanced to support automatic activation of distributed objects. This gives the CORBA functionality needed for our research in a simple form, integrated into our implementation language.

As with most RPC based systems, much of the code needed to send a message from one object to another is generated by a compiler that takes as input IDL specifications and gives as output source files in the target language. To translate from IDL into Modula-3, our compiler creates objects that derive from the Network Object (NetObi) base type. This introduces an additional exception, NetObj.Error, to all of the methods declared in an IDL file. This exception is thrown if any of the calls made cannot complete because of problems with the underlying object communication. Thus even though the methods declared for the event generator and event catcher objects return no values and produce no values that must be saved by the client of the calls, the methods are not asynchronous and return of control from such a call without the production of an exception indicates that the call succeeded.

When fed the interface definitions seen above, the IDL compiler will produce a Modula-3 interface for the types EventGen.T and EventCatcher.T. The task left to the programmer is to implement objects that support these interfaces. Implementations may add other functions that can be called locally. The implementations that follow provide objects that can be utilized by other objects within an address space to keep track of event class registrations. One of these objects will receive notifications, while the other keeps track of which notifications to send when an instance of an exported event class occurs.

The Modula-3 interface for the simple implementation of the event generator object is shown in Figure 3.

This interface extends that produced by the IDL compiler for our EventGen::T interface. This interface declares an opaque type T (referred to outside of this module as EventGenImpl.T)

that is a subtype of the type Public (referred to outside of this module as EventGenImpl.Public). An opaque type is one whose structure is revealed elsewhere; all we know of the type from this declaration is that it has at least all of the methods (and any other structure) defined for the Public type.

Type Public is declared as a subtype of the EventGen.T type, whose definition was produced by the IDL compiler. Objects of type EventGen.T must support the register and unregister methods. In addition to those methods, this interface defines three other methods that all objects that are of type EventGenImpl.Public (which includes objects that are of type EventGenImpl.T) must support.

The first of these new methods is addEvent, which simply tells the object taking care of event class registrations and notifications that the event identified with the indicated event class identifier is one that is exported by this object. The second method, delEvent, will end the ability of objects to register interest in a particular event class and will also keep any notifications of that event class's occurrence from being sent. The final method, trigger, tells the EventGenImpl.T object that an event with the indicated identifier has occurred, thus causing it

```
INTERFACE EventGenImpl;
IMPORT EventGen;
IMPORT VantageID;
IMPORT VantageObj;
IMPORT EventID;
TYPE T <: Public;
TYPE Public = EventGen.T
OBJECT
METHODS
   addEvent (
       newEvent : EventID.T);
   delEvent (
       delEvent : EventID.T);
   trigger (
       fireEvent : EventID.T)
   RAISES
   {EventGen.UnknownEvent};
END; (* object *)
PROCEDURE New (
   owner : VantageObj.T): T
   RAISES {};
END EventGenImpl.
```

Figure 3: Modula-3 interface for objects that generate event notifications

to send a notification to any object that has registered interest in that event class.

The interface also includes a New procedure, which creates an instance of the EventGenImpl.T object. This procedure can take an argument that indicates that the object created should consider itself part of a larger, compound object and should therefore return the object identifier of its owner when asked for its object id.

These additional methods are not available to clients who only know that the object is of type EventGen.T. They are part of the particular implementation. There is nothing about the EventGen.T interface that requires these functions. Other approaches to implementing the EventGen.T interface could use other functions.

It should be noted that the programming model used in this system is one of programming-language level objects being joined as aggregates to form a single distributed object. Thus when we speak of object identifiers, we speak of identifiers that are used to differentiate clusters of programming level objects. This is not a system in which there are "objects all the way down." Instead, objects within the programming language may be hidden as part of the implementation of objects as seen from the point of view of the distributed system.

The interface to the event catcher implementation is similar in that it declares an opaque type as a subtype of a public, abstract type that in turn inherits from the type that is generated from the IDL interface EventCatcher::T. The resulting interface is shown in Figure 4.

The EventCatcherImpl.T type will implement all of the methods of both the EventCatcher.T type (which includes only the notify method) and the EventCatcherImpl.Public type. This latter type includes two methods, register and unregister.

As with the simple implementation of the EventGen.T object, this implementation of the EventCatcher.T object is meant to provide functionality to a larger cluster of objects that wishes to receive notification of events from other objects. So like the EventGenImpl.T interface, there is a New function that takes as an argument the containing local object.

The register method allows local objects that are using the service provided by the Event-CatcherImpl.T object to tell the EventCatcher-

Impl.T that the object wishes to register interest in some event class occurring in some third object. This method requires that the object of interest be indicated, both by a handle that can be used to call the object and by its object identifier, and that the event class of interest be identified. In addition, this method requires the caller to pass in a closure; this closure includes the procedure to be called on receipt of notification, along with a structure that contains data to be used by that procedure.

On receipt of a register call, the EventCatcher-Impl.T object will register interest in the indicated event class with the indicated object. It will also set

```
INTERFACE EventCatcherImpl;
 IMPORT EventCatcher;
 IMPORT EventGen;
 IMPORT EvClosure;
 IMPORT NetObj;
 IMPORT VantageID;
 IMPORT VantageObj;
 IMPORT EventID;
 TYPE T <: Public;
 TYPE Public = EventCatcher.T
                OBJECT
METHODS
 register(
     objOfInterest :
 EventGen.T;
    objOfInterID :
 VantageID.T;
    eventOfInter : EventID.T;
     onNotify : EvClosure.T
 ) RAISES {
    EventGen.UnknownEvent,
    NetObj.Error};
 unregister (
    objOfInter : VantageID.T;
     eventOfInter : EventID.T;
 ) RAISES {
    EventGen.UnknownEvent,
    EventGen.NotRegistered,
    NetObj.Error};
 END; (* object *)
 PROCEDURE New (
    owner : VantageObj.T): T
    RAISES {};
END EventCatcherImpl.
Figure 4: Modula-3 interface for objects
that receive event notifications
```

up internal structures to insure that a thread will be spawned on receipt of a notification of that event that will run the closure handed in. Exceptions returned to any of these calls are passed along to the caller.

Similarly, the unregister method will call the unregister method in the EventGen.T object that was originally called during registration. Any exceptions raised during the attempt to cancel the registration of interest will be passed along to be handled by the local caller.

Implementation Results

Implementation of the simple EventGenImpl.T and EventCatcherImpl.T object types took 1,078 lines of Modula-3 code (including comments), of which 604 had to be written by hand and 474 were generated either by the IDL to Modula-3 compiler (450 lines) or by expansions of Modula-3 generic object types (24 lines). Tests were run on a SPARCstation 10 with a single 36 MHz. processor and 96 megabytes of memory running SunOS 4.1.3.

The test program created an EventGen.T object that exported 10 event class identifiers and an EventCatcher.T object. The test consisted of ten iterations of the EventCatcher.T registering interest in each of the event classes, triggering each of the event classes (and thus triggering a notification) in the EventGen.T, and then canceling the registration of interest in each event class. Times for the methods were kept separately. The closure registered as the notification handler simply returns without doing anything; however, a separate user-level thread is spawned for each closure.

The test was run in two configurations. In the first, both the EventGen.T object and the EventCatcher.T object were in the same process. In the second, the objects were in different processes on the same machine. Each test was run multiple times, but variations in the results from different runs of the same test were too small to be significant.

When both objects were in the same address space, 100 event class interest registrations took .045 seconds; 100 triggers, notifications, and notification handlers took .21 seconds; and 100 cancellations of interest took .016 seconds.

When the two objects were in separate address spaces the times, as expected, were considerably slower. In this configuration, 100 registrations took .53 seconds; 100 triggers, notifications, and

notification handlers took 1.04 seconds; and 100 cancellations of interest took .48 seconds.

These timings, and the small amount of code needed to implement the objects involved, give us reason to believe that we have met the goals stated earlier of defining a system that is both easy to implement and efficient.

Limitations of the Simple Approach

The simple approach to events and notifications meets a number of the goals we set out earlier for an event and notification system based on RPC. The way in which event classes are identified keeps the system from violating the abstraction boundary of an object, and keeps us from having to introduce a universal namespace for identifying event classes. As the sample implementation shows, the system can be implemented in a way that is cheap in terms of implemenation effort and runtime costs.

The simple approach does have some serious limitations. The CORBA approach to distributed object computing includes a model of objects that allows them to be active (currently loaded into a process and having at least one thread of control) or inactive (not associated with any process or thread, but with any persistent state on some form of stable store). Further, when a call is directed to an object that is inactive, the system will activate the object. We follow this model. However, activation of an object is a heavyweight activity. For notification of some events activation might be justified. However, there are other circumstances in which the delivery of a notification is not time critical and should not cause an activation; the notification should be postponed until the object is activated to service some other call. In our simple approach to notification delivery, there is no way for the system to allow this kind of "lazy" delivery.

Another limitation of the simple approach is the delivery guarantees that can be made for a notification. Currently there are none, and an object that is unable to deliver a notification to another object is left to deal with the failure itself. There is no way for the recipient to indicate that it desires some level of guarantee, nor is there an easy way to provide such delivery guarantees when they are requested.

Rather than change the basic protocol to deal with these problems, we address them by introducing a variety of third-party agents into the system. This is in keeping with our design goal of building complex features on top of a simple base. These agents must also meet the goal that their use should

be transparent to those who are not directly involved in using the service. We will now turn to the design of some of those services.

Notification Storage

One way of dealing with the problem of objects being activated to handle notifications that don't warrant that amount of effort is to interpose a notification storage agent between the object generating the notification and the object receiving the notification. This storage object can then implement various policies concerning when to pass the notification on to the object that originally expressed interest in the event class.

Such a notification mailbox would need to support the interface defined in "EventCatcher.idl." Once it did so, however, that object would look like any other event catcher object to an object that sent notifications. This is in keeping with our earlier goal of allowing complex functionality to be introduced in a fashion that is totally transparent to those not directly using the service.

One such notification storage object that we have already defined and implemented is a specialization of the EventCatcher::T object. This object is defined by the IDL interface shown in Figure 5.

Since this interface describes a type that is a subtype of the EventCatcher::T type, an Event-Box::T object will look just like any other Event-Catcher::T to an EventGen::T object.

The additional methods defined in the interface allow an object (itself an EventCatcher::T) to tell the EventBox::T that it wishes to have its notifications stored. The first method, request-Hold, initiates such a notification storage. The EventBox::T object is told the object identifier of the object that is requesting the storage, the event class identifier of the notification to be stored, and the object identifier of the object from which the notification will originate. In addition, the method requires that the requestor indicate to whom the notification should be forwarded, and the maximum number of these notifications to hold.

An object wishing to use this service will need to make two calls to register interest in an event class. The first call will be made to the Event-Box::T object's requestHold method, asking it to store notifications of a particular event class from a particular object. The second call will be to the register method in the object that will generate the notification. This call will be made with the

EventBox::T object being supplied as the destination of the notifications. This was the reason that call distinguishes between the object that is interested in the event class's occurrence and the object that is to receive the notification. Since there may be other third-parties in the chain, we also allow this distinction to be made in the call to the request-Hold method in the EventBox.

The calls to the two objects could be made in the opposite order. However, such an ordering makes it possible that a notification could be sent to the EventBox::T object before the requestHold asking for storage of that notification was complete.

```
#include "EventCatcher.idl"
module EventBox {
exception UnknownClient{};
exception
EventNotRegistered{};
exception
   NotifierNotRegistered{};
interface T :
EventCatcher::T {
void
requestHold(
   in VantageID::T holdFor,
   in EventID:: T eventID,
   in VantageID:: T holdFrom,
   in EventCatcher::T
                  forwardTo,
   in long maxToHold
   );
void
cancelHold(
   in VantageID::T holdFor,
   in EventID:: T eventID,
   in VantageID::T holdFrom
   ) raises (UnknownClient,
       EventNotRegistered
       );
void
emptyBox(
   in VantageID::T boxOwner
   ) raises (UnknownClient);
   };
};
```

Figure 5: IDL interface to an event notification mailbox

The cancelHold method simply informs an EventBox::T object that its services are no longer desired for a combination of client, event class, and event producer. This method will raise an exception if the client is unknown to the Event-Box::T object, or if the client has not requested that the indicated event class notifications be held.

The final method, emptyBox, simply causes all of the stored notifications being held for the object with the indicated object identifier to be delivered. This method will raise an exception if the object identifier is unknown to the Event-Box::T object.

This storage object implements a fairly simple form of notification storage and delivery. Like a bank of mailboxes, this sort of EventBox::T will store notifications. An individual client can ask for its mailbox to be emptied, at which time all of the notifications stored on its behalf will be delivered. The EventBox::T will then continue to store new notifications for that client until it asks for its mailbox to be emptied again.

There are other possibilities for delivery schemes. For example, the storage object could have a call that turned on delivery, causing all held notifications to be delivered but also delivering any new notifications immediately until another method was called, telling the storage object to return to a mode where it holds notifications. These different options will only be visible to the object that is asking for its notifications to be stored, and never to the object that is sending the notification.

Implementation of this notification storage type required another 1,131 lines of Modula-3, 688 lines of which had to be written by the programmer and the remainder of which were written by the IDL compiler or by expansion of generic types. As before, this number includes comment and blank lines.

Other Third-Party Services

While the notification storage server is the only third-party service that we have constructed so far, it is easy to think of other services that could be inserted transparently into the stream of event notifications in a similar way.

Perhaps the most obvious of these is a notification store-and-forward service that could relieve the object generating event notifications of the responsibility of dealing with failures in sending those notifications. Such a service would need to support the interface EventCatcher::T, enhanced to allow clients of the service to register information concerning the recipients of notifications and to specify reliability guarantees and other policy issues.

An implementation of such a service could receive a notification, attempt to pass it on to the intended recipient, and if that notification failed back off for some period of time and try to send the notification again. The time between notification attempts could increase each time the notification failed, until such a time as the service simply gives up, perhaps informing the object for which it was sending the notification.

Another service that could be provided would be an object that acted as a dispatcher for other objects that generate notifications. Rather than keeping track of all of the objects that have registered interest in getting a notification of some event class, an object could pass responsibility for this task to some third-party object. When some event occurs, only the third-party would need to be informed by the object in which the event occurred. All other objects would be sent a notification from the dispatching object. Indeed, such an agent could also supply store-and-forward functionality as outlined above.

One could easily imagine a distributed system in which each machine on the system had a single, well-known notification dispatcher (perhaps implementing some store-and-forward policy) and a well-known notification storage service. Such a system would have all notification messages go from the object in which the event occurred to the local dispatcher, from that dispatcher to the various notification storage services, and from those services to the (again local) objects that had originally registered interest. Such a design would isolate the network traffic for notifications to be between the specialized services.

Yet another service is one that can track the occurrence of events in objects and generate new events in response. Such an event monitor object could be made arbitrarily complex, generating different event classes for various sequences of events in different objects.

Other sorts of third-party services can be postulated; this is left as an exercise to the reader. What is important is that these services can be introduced so that objects using the basic protocols are unaware of the existence of those objects. This allows new services to be introduced without changing the objects that are not direct clients of (but may, perhaps, be recipients of) the service.

Remaining Issues

While the system described in this paper is powerful and flexible, there are still some open issues concerning the use of events and notifications in an RPC based distributed system.

Perhaps the simplest of these is the question of whether notifications should carry sequence numbers. Currently, an object receiving multiple notifications of an event class from an object has no way of knowing the order of those notifications. However, since notifications carry no information other than that the event occurred, there is no need to order the notifications received, for there is no difference between the notification for the nth occurrence of an event class and the notification of some other occurrence of that event class.

Sequencing might be an issue when notifications are held in a notifications storage server, however. Since a server like the one we have implemented is told a maximum number of notifications to hold, it is possible that an object will miss some notifications because the server has discarded them. Adding a sequence number to the information passed by a notification would allow the client of such a notifications storage server to determine how many notifications were discarded.

We believe that such functionality, if necessary, should be added to the notification storage service rather than to the basic notification interface. However, future experience may change this view.

This does not address the detection of duplicate notifications, which is another reason for introducing sequence numbers. If such detection is needed, we can easily add a way of delivering an event sequence number to the protocol.

A second issue has to do with cleaning up registrations of interest when the object that made the registration disappears (for whatever reason) without cancelling that registration. The current implementation will not find this out. The best that can be done is that the object sending the notification can find out that the object is unreachable. This, however, could be due to repeated bad luck with the network. Just retaining the registration means that over time the amount of "orphaned" registrations could grow without bound.

A final issue is a more theoretical concern. One general principle of object-oriented programming is that interfaces not related by inheritance should be independent at least to the extent that an object may support one interface without supporting the other. Yet our approach to events and notifications appears to violate this principle. An object that supports only the EventGen::T interface but does not support any interface that contains a method that returns an event class identifier does not seem very interesting. In the same way, it would seem that any object that supports an interface that contains a method that returns an event class identifier should support the EventGen::T interface.

There are a number of unsatisfying approaches to this issue, including requiring any interface that contains a method returning an event class identifier to derive from the EventGen::T interface. We are more inclined to the view that this kind of dependency between interfaces shows that there is more to interface relationships than can be captured in an inheritance hierarchy.

Conclusions

We have shown how a system of events and notifications can be built on top of simple interfaces in a distributed system built on the paradigm of RPC for communication. The interfaces used are very simple, allowing cheap implementations that are efficient.

We have also shown how more complex functionality can be introduced into the simple system by introducing third-party servers. These servers can be added to the system in such a way that those who merely receive their services (as opposed to those who make use of those services) need not be aware of them. Further, the addition of such services can be made in a way that does not add any complexity to the basic interfaces.

A number of lessons can be learned from this exercise. The first is that RPC and event notifications can co-exist in a distributed system, especially if one is careful to make sure that each does only the work for which it is best suited. Another is that changes in abstract state can be exported without violating the object metaphor. Finally, we have demonstrated how complex services can be built on top of simple interfaces by introducing third-party objects that support the complexity.

References

- [1] Babaoglu, Ozalp and Keith Marzullo. "Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms" in Sape Mullender (ed.), *Distributed Systems*, *Second Edition*, Addison-Wesley (1993).
- [2] Birman, K.P. and T.A. Joseph. "Exploiting Virtual Synchrony in Distributed Systems," in *Proceedings of the Eleventh Symposium on Operating Systems Principles*, Austin, Tx. (1987).
- [3] Birrell, A. D. and B. J. Nelson. "Implementing Remote Procedure Calls." *ACM Transactions on Computer Systems* 2 (1978).
- [4] Birrell, Andrew, Greg Nelson, Susan Owicki, and Edward Wobber, "Network Objects," Digital Equipment Corporation Systems Research Center Technical Report 115 (1994).
- [5] Dasgupta, P., R. J. Leblanc, and E. Spafford. "The Clouds Project: Designing and Implementing a Fault Tolerant Distributed Operating System." *Georgia Institute of Technology Technical Report GIT-ICS-85/29* (1985).
- [6] DellaFera, C. Anthony, Mark W. Eichin, Robert S. French, David C. Jedlinsky, John T. Kohl, and William E. Sommerfeld, "The Zepher Notification Service," *Proceedings of the Winter USENIX Conference* (1988).
- [7] Edelson, Daniel. Enterprise Wide Distributed Programming with InterStage: An Overview, talk presented at the 1994 USENIX C++Advanced Topics Workshop, Boston, MA. (1994).
- [8] Hutchinson, N. C., L. L. Peterson, M. B. Abott, and S. O'Malley. "RPC in the x-Kernel: Evaluating New Design Techniques." *Proceedings of the Twelfth Symposium on Operating Systems Principles* 23, no. 5 (1989).
- [9] Khalidi, Yousef A. and Michael N. Nelson. "An Implementation of UNIX on an Object-Oriented Operating System." *Proceedings of the Winter USENIX Conference* (1993). Also *Sun Microsystems Laboratories*, *Inc. Technical Report SMLI TR-92-3* (December 1992).
- [10] Nelson, Greg (ed.), Systems Programming with Modula-3, Prentice Hall (1991).
- [11] The Object Management Group. "Common Object Request Broker: Architecture and Specification." *OMG Document Number* 91.12.1 (1991).

[12] Parrington, Graham D. "Reliable Distributed Programming in C++: The Arjuna Approach." *USENIX* 1990 C++ Conference Proceedings (1991).

[13] Shirley, J, A Guide to Writing DCE Applications, O'Reilly & Associates (1992).

[14] Skeen, Dale. "An Information Bus Architecture for Large-Scale, Decision-Support Environments", *Proceedings of the Winter USENIX Conference* (1992).

[15] Zahn, L., T. Dineen, P. Leach, E. Martin, N. Mishkin, J. Pato, and G. Wyant. *Network Computing Architecture*. Prentice Hall (1990).

Jim Waldo (jim.waldo@east.sun.com) is a Senior Staff Engineer at Sun Microsystems Laboratories, East Coast Division, working in the area of reliable large-scale distributed systems. Prior to joining Sun, he worked in distributed systems and object-oriented software development at Apollo Computer (later Hewlett Packard), where he was one of the original designers of what has become the Common Object Request Broker Architecture (CORBA).

Ann Wollrath (ann.wollrath@east.sun.com) is a Member of Technical Staff at Sun Microsystems Laboratories, East Coast Division, working in the area of reliable large-scale distributed systems. Prior to joining Sun, she worked in the Parallel Computing Group at the MITRE Corporation investigating optimistic execution schemes for parallelizing sequential object oriented programs.

Geoff Wyant (geoff.wyant@east.sun.com) is a Staff Engineer at Sun Microsystems Laboratories, East Coast Division, working in the area of reliable large-scale distributed systems. Prior to joining Sun, he worked at CenterLine Software and Apollo Computer (later Hewlett Packard), where he was involved in the original design and implementation of the Network Computing System.

Samuel C. Kendall (sam.kendall@east.sun.com) is a Member of Technical Staff at Sun Microsystems Laboratories, East Coast Division, working in the area of reliable large-scale distributed systems. Before joining Sun, Sam worked on C and C++ programming environments at CenterLine Software, and on the first compiler for C*, a data-parallel cousin of C++, at Thinking Machines Corporation.

POTPOURRI II

Session Chair: Lori Grob, Chorus Systèmes

NOTES

Turning the AIX Operating System into an MP-capable OS

Jacques Talbot Bull

Abstract

This paper describes those MP features that Bull and IBM together introduced into the AIX operating system to support the Symmetric Multiprocessor machine marketed by Bull under the Escala name and by IBM under the RS/6000 Models G30, J30 and R30 names. The PowerPC architecture and the AIX operating system present some specific challenges. We present the major problems encountered and how they were solved.

1. Introduction

This paper is quite broad in the sense that changing a sophisticated UNIX implementation such as the AIX operating system into an MP-capable system is a complex task. So we have tried to briefly describe the problems encountered and then to provide a more detailed description of some of them, which were specific to the AIX operating system and the PowerPC.

2. The Hardware Architecture: a summary

Only the HW features which have a significant impact on the SW will be described here. The architecture is a so-called SMP, Symmetric Multi Processor based on PowerPC [IBM94]. An SMP is a system where several CPUs share a common memory address space and I/O subsystem. There is no memory local to CPUs (except caches which are almost invisible to SW). The Symmetric attribute is perhaps not the most significant one since most machines in the MP category to-day are symmetric, in the sense that all processors get equal access to all I/Os. The exceptions to this rule are primarily low-end Intel-based MP servers derived from PC technology or machines dedicated to specific needs (e.g. real-time).

So SMP is better described as "Shared memory with HW coherency". It means that all CPUs see the "good old programming model" of a single memory with a more or less uniform access time, i.e. not depending on the addresses accessed. Moreover, all CPUs, when reading the "same" address, get the same value. In presence of copy-back caches, it is not so easy to achieve this simple property; this is the purpose of the HW cache coherence protocol, named according to the MESI acronym which designates the 4 possible states of a cacheline in the state machine (Modified, Exclusive, Shared, Invalid). The SW can safely ignore this complexity, assuming that the HW is properly functioning.

The same characteristics of uniform access time and coherent memory allow to run a single copy of the operating system on the machine. So all CPUs can share a single queue of ready-to-run execution threads, resulting into better system load balancing. Getting applications for the platform is pretty simple. There is actually no porting activity in the general case, since applications can rely on the traditional assumptions that they have made since computers are with us. Binary compatibility with the uniprocessor applications is the rule.

In contrast, architectures such as MPP (Massively Parallel Processors) and other kinds of tightly coupled clusters, where the above two characteristics are not present, generally require that applications have some knowledge of the HW architecture.

2.1 PowerScale architecture

This specific SMP architecture is called PowerScale. There are up to 8 PowerPC (601, 604 or 620) CPUs accessing a common memory through a cross-bar for data and a bus for addresses. A cross-bar is a data exchange device modeled after a telephone cross-bar switch where each agent has a private data path to the interconnection HW and multiple connections can occur simultaneously. An I/O bridge allows access to two MCA busses.

The PowerScale architecture is optimal from a priceperformance standpoint for the target we were shooting at, i.e. a mid-range SMP optimized for 4 CPUs, still efficient at 8 CPUs and sellable in monoprocessor.

2.2 Caches

Each of the 8 PowerPC CPUs has a (relatively) small internal (32 or 64 Kbyte) Level-1 cache and a large external (1 or 2 MByte) Level-2 cache. Each cache is organized in "lines". Data transfers and coherency updates take place in unit of lines. A cache line is 32 or 64 bytes wide, depending on the PowerPC model. Both caches are copy-back (sometimes called write-back), meaning that when a write is performed, the modified value is written only in the cache, and pushed in lower memory hierarchy levels only when required, typically when the "dirty" cache line is needed for other data.

The other option is write-through, where main memory is updated on each write. This technique is much simpler but considerably less efficient.

In a copy-back cache, when several copies of a "clean" (unmodified) line exist in several caches, and one of these is modified, there are two options for other copies: update or invalidate. Write-invalidate is best from a performance standpoint for data which will probably not be used in the near future by the CPUs where the line is invalidated. Write-update is best when the probability of reuse is high. In most cases, for UNIX applications and kernel pages, write-invalidate is the preferred mode since most lateral misses (cache to cache) are related to a working set migration which is a one-way event and seldom associated with real data sharing. Since the PowerPC architecture implements only write-invalidate, there is anyway no choice to make.

It is possible, HW-wise, to tag pages as cached (copy-back or write-through) or uncached. Except for some phases of bootstrap in AIX, all memory is tagged as cached and copy-back.

2.3 Atomic operations

To construct the locks that SMP SW needs, an atomic operation is needed. It is not the traditional test_and_set or compare_and_swap instructions of CISC processors that are used. The PowerPC, like many RISC architectures, implements load_and_reserve and store_conditional instructions such that if a reservation bit is "broken" between the Load and the Store operations, the Store fails. These two instructions allow the efficient implementation of all the atomic operations that concurrent SW needs, including test_and_set and compare_and_swap.

2.4 Memory order model

Until recently, UNIX multiprocessors had a *strongly* ordered memory model. This means that the order of loads and stores as seen by the program cannot be changed between the output of a CPU and the memory

system, where the other CPUs see the effect of these operations. Modern architectures are weakly ordered. The HW is allowed to reorder loads and stores between the CPU and the memory. This is done to optimize performance, when the first operation has to go to main memory for completion while the second one can be rapidly globally performed in the Level-1 cache. In that case, the second is globally performed before the first. Of course, the order of operations as seen by one CPU is not changed, because the HW keeps track of data dependencies and always provides the latest value of a data to the CPU, even if not yet globally performed. Only the visibility of these operations from the other CPUs is impacted. This has some importance only when CPUs synchronize communicate with one another, because one CPU could get stale values for some data.

A similar issue results from aggressive prefetching of data (*speculative execution* found on 604 and 620) which allows a CPU to read some data very much in advance, and even across a test-and-branch sequence. This again allows a CPU to read data which has stale value.

Special instructions (*isync* and *sync*), also called import and export fences, allow to solve these problems. However, inserting these instructions stalls the pipelines and is therefore a performance drain.

2.5 Interrupt handling

The HW allows interrupts from one bus adapter to be addressed either to a specific CPU or to a set of CPUs. This set actually designates "one among all CPUs" and this CPU is dynamically chosen by the HW at each interrupt from a set of registers, one register associated with each CPU. Each CPU writes in this register a value indicating its willingness to receive interrupts.

3. Some high-level SW issues

Let's now jump abruptly to a very different level of abstraction. The main issue to transform a UniProcessor (UP) Operating System like the AIX operating system Version 3 into an MP Operating System like AIX Version 4 is to protect the coherency of the data structures managed by the kernel.

This is because several CPUs can simultaneously manipulate these data structures. The conflict can be a thread-thread conflict or a thread-interrupt conflict. On a UP machine, the thread-thread conflict cannot exist since only one flow of execution exists at any time (true only for a non-preemptable kernel); the thread-interrupt conflict is solved by masking the processor against interrupts. In an MP OS, locks must be added.

Apart from becoming an MP OS, the AIX operating

system was also turned from a process-based kernel to a thread-based kernel. Several threads of execution exist within the scope of a process which is now a passive entity representing an address space and access rights. Threads are especially useful for MP architectures since they enable finer-grain parallel programming, compared to current multi-process applications; this gives an opportunity to have a better speed-up on multi-threaded applications, as opposed to the scale-up (more throughput) which can be obtained without parallel programming. A single multithreaded application will run faster for a single user on an SMP machine.

Threads-based programming requires a new API (POSIX 1003.1c) and thread-safe libraries. It opens the way to portable thread-based applications. We will not describe threads in more detail in this paper.

Turning a UP UNIX kernel into an MP kernel has been described several times during the last years in various USENIX proceedings [Cam91] [Car93] [Pow91]. We will therefore try to focus below on AIX-specific or PowerPC-specific issues.

4. Locks

4.1 Some interesting AIX Version 3 features

AIX Version 3 has two specific properties which affect the MP process.

• preemptability: when an interrupt occurs that makes schedulable a thread which has a higher priority than the currently running thread(s), a context switch is required. If the current thread is running in user mode, there is no issue. It is temporarily suspended, and this process is called preemption. However, if the current thread is running in kernel mode, most UNIXes do not allow the preemption to .occur. This is to prevent corruption of the kernel data structures. The context switch is delayed until the current thread goes back to user mode. Since this can last quite a while on long kernel operations, this behavior is detrimental to response time and real time requirements.

There are two possibilities to overcome this: preemption points and full preemptability. Preemption points are the poor man's solution, meaning that the kernel is not preemptable except at some specific points. Full preemptability means that the kernel is always preemptable except during a few code sequences. AIX Version 3 is fully preemptable. This implies a locking scheme to

maintain data coherency.

• pageability: the AIX kernel code and data structures are pageable, except for some parts pinned in memory to avoid deadlock scenarios. This minimizes the amount of real physical memory required by the kernel by paging-out seldom used code and data. It allows to overdimension kernel tables so that the hideous error message "too many processes or messages or whatever" encountered on traditional UNIXes is never seen on AIX. When a demand peak for a resource occurs, one or more pages of the corresponding table are paged-in and used. When no longer required, they are paged-out.

So page faults and their associated context switches can be encountered at any point in time when kernel code runs. This is another case of unexpected kernel preemption.

To implement the two properties above, AIX Version 3 has a simple locking scheme with a coarse granularity; this scheme cannot be used for an MP system because the coarse lock granularity results in a very high contention rate on the few locks and a very poor overall scalability versus the number of processors. Processors spend their time waiting for the locked resources. However, the V3 locking scheme provided pointers to the critical areas that needed to be addressed.

These two AIX features obviously had to be kept for the MP version. It should be noted that the refinement of the locking scheme adopted for the sake of MP efficiency had the side effect of improving the preemptability of the kernel. In fact, tuning the locks for low contention tends to reduce the pathlength of the code spent with preemption off.

4.2 What should be locked?

Two basic approaches to locking exist:

- code-based locks are coarse grain locks which encapsulate large pieces of code and associated data. Basically, when entering a subsystem, a lock is taken and released only at subsystem exit. This is simple but collision rates can be high since such locks are taken for a long time.
- data-based locks are associated with data structures, or part of them, and lock-unlock operations are inserted in the code so that data protected by the lock are kept coherent when the lock is free. Also data-based locks can be coarsegrained (a whole table), medium grained (some elements of a table) or fine grained (one element of a table, or even a single item in a structure).

Both types of locks are used in AIX Version 4. The choice of lock type is dependent upon how critical to performance is a given subsystem. For example, the CD-ROM filesystem has few requirements for simultaneous access due to the physical constraints of the device. So a code-lock can be used.

On the other hand, the table of messages for System V IPC must be fine-grained because this is critical to the aggregate message switching capabilities of the system, in presence of several message queues, readers and writers.

4.3 Which type of locks?

Many lock semantics can be imagined. Because we wanted to be able to borrow some code from the OSF/1 kernel, and because of the experience Bull had accumulated with the MP aspects of OSF/1, we decided to model our locks on the OSF/1 kernel locking scheme.

In the case of a resource being unavailable, the requester thread blocks. This can be either by spinning (sometimes called "busy wait") or by releasing the CPU and going to sleep. Sleeping is not allowed if called from an interrupt level because it leads to deadlock situations.

Two types of locks are used:

- simple locks: as the name implies, this is the most rudimentary form and as a result the most efficient and frequently used. The lock is a simple word and if busy, the requester spins. However, the lock implementation has been optimized as follows:
 - After some amount of spinning, the requester of a busy lock will sleep (except if the request is made from a interrupt-level or masked against interrupts). The appropriate spinning time is approximately the time spent to execute a process switch.
 - If a lock is busy and the owner of the lock is not running on any of the CPUs, the requester goes to sleep. This is because the probability that the lock is freed in the spinning time-frame is almost zero.
 - The owner of a lock is not running either because it is runnable but not currently running on a processor, or because it is waiting for some event (e.g. page fault resolution).
- complex locks: these are multiple-readers, single-writer semaphores. They can optionally be tagged as recursive. Recursivity is generally discouraged in the locking scheme to keep it simple and deadlock free. However, to be able to import recursive code, this option is left open to the

programmer. Threads requesting a busy complex lock spin and then sleep just like requesters of simple locks.

They are used only if Reader-Writer semantics make sense (e.g. in the filesystem) or if recursivity is needed.

Both types of locks can create the priority-inversion syndrome. This happens when high priority threads wait for a resource held by a lower priority thread. It results in a degradation of the response time. To overcome this, the priority of the thread owning a lock is temporarily raised to the priority of the highest priority thread waiting for the lock. This is sometimes called "priority promotion".

4.4 How many locks, how often?

A system may have many or few locks. It may use them often or seldom. In a first approach, fine-grain parallelization implies many locks, taken often. Coarse-grain parallelization has few locks seldom taken. Obviously, there is a continuum in a two-dimensional space. It is not always obvious to characterize a locking scheme.

At first sight, it looks that fine-grain locking is always better. It results in very short time spent while holding a lock, so low probability for collision (finding a lock busy). However, locking does not come for free. As opposed to older architectures where writing a word and accessing a lock through a test_and_set instruction were in the same ballpark in terms of execution time, there are orders of magnitude of difference between reading or writing a word in Level-1 cache (1 cycle) and taking a lock (around 50 cycles in the best case). The high number of cycles associated to locking is due to several factors, in order of importance:

- need to synchronize the pipelines on lock operations
- cache misses: processors are much faster these days, but memory does not keep up.
- SW overhead: for various reasons and despite all the tuning, several bookkeeping instructions are needed around the atomic core

So too much locking and unlocking is detrimental to performance. Instead of taking a lock for the minimal amount of time needed from a data structure coherency standpoint, the programmer has to consider whether the lock will not be needed again soon, and keep the lock during this interval, even if not strictly needed. The same applies to whether it is better to have several locks for a set of data structures and play subtle games or have a single lock held for a longer time but toggled less often.

So locking the kernel is finding a trade-off between too few locks and too much contention or too many locks and too much overhead. In both cases, performance suffers. There is an optimum but finding it is a matter of calculations, experience and finally trial-and-error process [Cam91].

Some figures below give an idea of the locking scheme complexity in AIX version 4. They must be considered as orders of magnitude more than exact figures. There are around 200 classes (types) of locks. The actual number of instances of locks depends on the size of various tables and is not very significant. The number of lock operations (statically counted) is around 500. The number of locks operations performed per second on a timesharing workload (SPEC SDET) is around 10,000 lock-unlock pairs per second, on a quad machine with 75MHz 601 processors.

4.5 How to avoid locks

Some parts of the kernel fit into well defined frameworks and in that case the burden of locking can be delegated to the framework code.

- Drivers which have no strong performance requirements can be "funneled". In that case the drivers framework locks the driver on a "master" processor and all accesses are serialized via a single lock. This scheme is used only for third party-devices. None of the Bull or IBM-developed drivers use it.
- The streams framework [Rit84], taken from OSF/1, can provide locking at various levels, described below from the highest level of concurrency to the the lowest level:
 - queue level: only access to message queues is protected. Modules must provide their own locking for shared state or upstream downstream synchronization.
 - queue pair level: both upstream and downstream queues are protected. If some data are not "per-stream", i.e. they are shared between several streams, the module must take care of their locking. This is used by TTY line disciplines, which have only per-stream data.
 - module level: all queues and shared data associated with a module are protected. This is the default mode which can be safely used by most UP streams modules, which want to ignore MP issues.
- arbitrary level: allows to arbitrarily group together from the locking standpoint modules

- which share data other than via queues.
- global: one single lock for the whole streams subsystem, for debug purpose only

In AIX Version 4, the TTY subsystem, SNA and Netware stacks are implemented within the streams framework.

4.6 Deadlocks

As soon as we have more than one lock in a system, a potential for deadlock exists. One well-known strategy to avoid deadlocks is to implement a lock hierarchy, so that locks are always taken in the same order. We chose to implement a partial hierarchy, i.e. not global to the whole kernel but per subsystem. This hierarchy is defined in the registration process (see below under *Lock instrumentation*). However, it is not global nor enforced dynamically for reasons similar to the ones explained in [Pac91]:

- strict ordering is hard to enforce globally and not always necessary, if the programmer knows that the deadlock cannot occur for some reason.
- impact on lock code pathlength and therefore performance
- desire to be able to import code from OSF/1 without major restructuring

We chose instead to use a static deadlock analyzer, called SDLA (Static DeadLock Analyzer). This tool processes kernel code to detect potential deadlocks by exploring the locking scheme tree [Kor89]. It is derived from lint. SDLA does not use the static description of the lock hierarchy but constructs its own vision of the lock hierarchy while walking the tree. Two common problems found in many similar tools are the "noise" (false deadlocks) and the time taken to explore the tree and perform the analysis. We found several techniques to overcome these in a practical manner (patents pending) and we were able to discover several kernel design errors. We use the SDLA to continuously inspect the source tree to ensure that the addition of features and correction of bugs do not introduce new deadlocks.

We foresee an enhancement to SDLA to be able to detect underlocking. Underlocking happens when data normally protected by a lock are manipulated without the lock being held. This can lead to kernel data corruption and system crashes. It is the most critical area for an MP kernel, since deadlocks are less difficult to debug. This will be similar to WARLOCK [Ste93].

4.7 Lock instrumentation

A comprehensive lock instrumentation is key to the kernel tuning process [Car93]. A symbolic naming and registration scheme has been adopted for all locks. This facilitates lock designation by the analysis tools. Locks are named as:

Subsystem_name\$Lock_family_name\$Occurrence_number

We implemented two levels of lock instrumentation:

 One is almost always on during validation phases (selection at bootstrap time) and records the number of acquisitions, misses and sleeps. The overhead has to be as low as possible. The current implementation adds 10% to the lock-unlock pair execution time. A tool named *lockstat* allows the customer or field-service to analyze the behavior of kernel locks.

Below is a display of *lockstat* output, taken during the run of a database benchmark early during development.

The 20 locks with largest %Ref are shown:

requests

%Block is the % of blocking (spin or sleep) requests versus all requests for this lock.

%Sleep is the % of blocking requests resulting in sleeping versus all

requests for this lock.

-	Name	Ocn	Ref/s	%Ref	%Block	%Sleep
PROC	PROC_INT_CLASS	-1	8434	29.89	18.77	0.00
IOS	UPHYSIO_LOCK_CLASS	-1	2483	8.80	6.74	0.00
PMAP	PMAP	0	2108	7.47	1.60	0.00
DISK	SCDISK_LOCK_CLASS	-1	1495	5.30	4.22	0.00
IPC	SEM_LOCK_CLASS	2	1472	5.22	7.75	0.00
LOCKL	LOCKL	7	1035	3.67	0.00	0.00
IOS	IOS_IPOLL_CLASS	2	1030	3.65	0.48	0.00
PFS	JFS_LOCK_CLASS	-1	997	3.53	1.46	0.00
XLVM	LVM_LOCK_CLASS	0	996	3.53	3.24	0.00
IPC	SEM_LOCK_CLASS	2	569	2.02	4.12	0.00
IPC	SEM LOCK CLASS	2	524	1.86	3.73	0.00
IPC	SEM LOCK CLASS	2	523	1.86	3.90	0.00
VMM	VMM_LOCK_SCB	443	499	1.77	1.66	0.00
IPC	SEM_LOCK_CLASS	2	475	1.68	3.49	0.00
PROC	TRB LOCK CLASS	3	445	1.58	0.05	0.00
PROC	TRB LOCK CLASS	2	443	1.57	0.03	0.00
PROC	TRB LOCK CLASS	0	441	1.56	0.02	0.00
PROC	TRB LOCK CLASS	1	438	1.55	0.04	0.00
TCPKER	DEMUXER_LOCK_FAMILY	45172	417	1.48	3.08	0.00
PROC	TOD_LOCK_CLASS	-1	399	1.42	12.23	0.00

Subsys Name Ocn designate the lock.

Ref/s is the number of requests

for this lock per second.

%Ref is the % of requests for

this lock versus all lock

• For a more detailed lock analysis, we have a trace-based instrumentation, which can be turned on-off dynamically. We use the AIX trace tool and insert trace hooks in lock code. This allows us to compute the time spent under locks and get a comprehensive view of lock behavior. The LCA (Lock Contention Analyzer) is a Motif-based tool giving a complete picture of all lock parameters. It is used only in the development organization. Below is a display of LCA output, taken during the run of a database benchmark early during development.

The 20 locks with largest Coll Sec are shown:

CPU Avg is average CPU time (in microseconds) spent with the lock held during one locking (i.e. between one lock and the corresponding unlock)

CPU Max is maximum CPU time (in microseconds) spent with the lock held during one locking

The data above were taken during the development process and do not reflect the behavior of the shipped system. For example, at that point in time, %Sleep was always 0 because the locks were

TABLE Collision Section (times in 10 s)

		The same of the sa											- 10		
Class	1	Ocn	Туре	 	Count	1	Coll	Sec (%)	1	CPU (ms)	1	CPU Avg	1	CPU Max	
PROC_INT_CLASS	1	i	Simple	1	93579	1		6.108	1	2885.994	ı	30.84	i	1712.00	
SEM_LOCK_CLASS	1	2	Simple	1	15367	1		2.816	1	1330.436	1	86.58	1	2323.46	
UPHYSIO_LOCK_CLASS	1		Simple	1	25658	1		2.466	1	1165.270	1	45.42	1	1609.22	
SCDISK_LOCK_CLASS	1	1	Simple	1	15397	1		2.039	1	963.444	1	62.57	1	1336.19	
LVM_LOCK_CLASS	1	0	Simple	1	10265	1		1.552	1	733.498	1	71.46	1	576.38	
DEMUXER_LOCK_FAMILY	1	45172	Simple	1	4267	1		1.132	1	534.809	1	25.34	1	365.82	
IOS_IPOLL_CLASS	1	30	Simple	1	2580	1		1.114	1	526.462	1	04.05	1	546.30	
LOCKL	1	45	Lockl	1	13043	1		1.071	1	506.051	1	38.80	1	1790.21	
IOS_IPOLL_CLASS	1	27	Simple	1	2383	ł		1.049	1	495.487	1	07.93	1	545.28	
DEMUXER_LOCK_FAMILY	1	41076	Simple	1	3783	1		0.967	1	456.725	1	20.73	1	518.27	
SEM_LOCK_CLASS	1	2	Simple	1	5899	1		0.805	1	380.185	1	64.45	1	1300.61	
SEM_LOCK_CLASS	1	2	Simple	1	5231	1		0.719	1	339.934	1	64.98	1	1806.98	
SEM_LOCK_CLASS	1	2	Simple	1	5407	1		0.713	1	336.979	-1	62.32	1	1276.16	
DEMUXER_LOCK_FAMILY	1	61556	Simple	1	2104	1		0.673	1	318.065	1	51.17	1	309.89	
SEM_LOCK_CLASS	1	2	Simple	1	4800	1		0.654	1	309.150	1	64.41	1	1171.71	
DEMUXER_LOCK_FAMILY	1	116	Simple	1	1937	1		0.608	1	287.322	1	48.33	1	318.59	
VMM_LOCK_SCB	1	489	Simple	1	5234	1		0.559	1	264.332	1	50.50	1	387.33	
JFS_LOCK_CLASS	1		Simple	1	11736	1		0.373	1	176.423	١	15.03	-1	1192.45	
U_TIMER_CLASS	1	41	Simple	1	13639	1		0.366	1	173.051	1	12.69	1	581.76	
IOS_IPOLL_CLASS	1	2	Simple	1	11182	1		0.302	1	142.643	1	12.76	1	47.62	
	-+-	+		-+-		+-			+-		+		+		_

Class Ocn designate the lock is the lock type: simple or R/W Type or Lockl (AIX V3 backward compatibility) Count is the number of lock requests during the period is the collision section, i.e. the Coll Sec(%) % of CPU time when the lock is is total CPU time (in ms) spent CPU (ms) with the lock held during the period

tuned to spin for a very long time. A lock in the process management (PROC_INT_CLASS) was a point of contention. This is clear from the %Ref and %Block in the *lockstat* display or the Coll Sec(%) in the *LCA* display. This lock is both taken often (%Ref) and often missed (%Block). With this information, this problem was taken care of in the MP tuning process.

Precedence rules among locks in a subsystem are registered in a database. This allows the LCA tool to detect rule violations and issue warnings.

4.8 Debugging

The identification of the thread owning a lock is systematically registered so that the deadlock detector contained in the crash analyzer can provide extended information. This is true even in production kernels.

During the development phase, locks can be compiled with additional sanity checks turned on so that "asserts" can be used. They are however too costly to go into production.

The symbolic kernel debugger has been enhanced so that when a CPU encounters a breakpoint, the other CPUs are also stopped through the CPU to CPU communication channel (MPC, see below). It is then possible to switch the debugger from one CPU to another for CPU-specific data.

4.9 Testing

For each subsystem, we have a set of tests specifically targeted to lock stressing. With the *lockstat* tool, we are able to ensure that the collision rate is high enough on each and every lock so that we are sure that they are appropriately stressed, looking either for overlocking (\rightarrow deadlocks) or underlocking (\rightarrow race situations leading to data corruption). The collision rate is also randomly varied between maximum value and 0 to explore windows of vulnerability.

4.10 Kernel Implementation issues

The price of locks After carefully coding lock and unlock primitives for simple locks, in assembly language, it was determined that the lock-unlock pair price in the lock free case was around 100 cycles (on a PowerPC 601). Notice that the number of instructions is much smaller. Some lock instructions (e.g. *sync*) are costly in terms of cycles because the 601, as any sophisticated processor, needs to flush its read and write queues.

MP/UP overhead Locking and unlocking brings an overhead compared to a UP kernel. We took the goal to limit this overhead to 5% on the throughput of macro benchmarks like the TPC family or SPEC SDM and LADDIS.

The core of the AIX kernel (i.e. not counting the drivers and loadable kernel extensions like NFS) actually exists in two versions derived from a single source tree. In the MP version, all locks are enabled. In the UP version, only locks mandatory to allow pageability and preemptability are enabled using #ifdef. So, the UP systems do not incur the MP performance penalty. The appropriate kernel is automatically selected at bootstrap time.

Thundering herd problem This occurs when several threads are queued waiting for a resource, the resource is freed and several waiting threads are awakened at the same time. For simple locks, this is avoided by selectively waking only the highest priority sleeping thread. For complex RW locks, the highest priority writer is awakened, or all the readers if no writer is waiting.

Interrupts and locks On an MP system, to protect thread-interrupt critical sections, it is necessary to take both a lock (for protection against other CPUs) and to mask interrupts appropriately (for protection against your own CPU). For performance and readability reasons, we packaged these operations together into two primitives: disable_lock() and unlock enable().

AIX Version 3 already had two levels for interrupt handling. Part of the interrupt code is executed at the HW interruption level. When HW critical management has been completed, the bulk of the interrupt processing, if necessary, is handled "off-level" at a lower SW interrupt level.

Since the AIX Version 4 kernel is thread-based, there was an opportunity to transform interrupt handling into normal thread-level code [Pow91]. We decided not to do so because of the performance overhead of thread level handling versus off-level interrupt handling.

4.11 Tuning the lock system

Because we had a previous experience in MP UNIXes, we decided to shoot directly for a 4-way MP-efficient for the first release. A number of guidelines were given to developers. The *collision section* of a lock is the % of the total CPU time spent under this lock in the uniprocessor case. The objective was that no collision section should be greater than 2.5%, for significant benchmarks (e.g. TPC-C, SDET, LADDIS). This can be measured with the LCA tool on a UP system. So even without any SMP HW available, we were able to determine early during the development process all the potential granularity problems and established a list of "locks to be broken". We also identified too fine-grained locks which had to be coalesced.

We believe that the scheme put in place can scale up to 8-way without major redesign, requiring only a series of measure-and-tune cycles that we will implement between the first and the second MP AIX releases.

4.12 User mode locks

Some applications and most importantly database managers implement locks in user-mode because they cannot afford the performance overhead of system provided semaphores (System V IPC) and started implementing multithreaded applications before a standardized API was available. These locks are often called latches.

On a UP AIX 3.2, these locks are implemented with a *compare_and_swap()* library routine which uses the *load_and_reserve* and *store_conditional* instructions on PowerPC machines or a system call on POWER machines, where atomic instructions are not available.

On MP machines, this scheme does not work because of the weakly ordered memory model described in the HW section. As the insertion of the fence instructions is a delicate process and moreover because these fence instructions depend on the type of PowerPC processor (601, 604 or 620) for optimal performance, we provided a simple API to hide this complexity:

_clear_lock() issues the appropriate export fence and clears the lock-word.

Programmers can then combine these to implement the appropriate type of lock for their application.

An alternative is to use the POSIX1003.1c mutexes and condition variables provided with the pthreads library.

5. Other atomic operations

The availability of *load_and_reserve* and *store_conditional* instructions enables the construction of other atomic operations. They provide better performance when compared to locks for certain operations. They usually do not need fences because no data other than the parameter of the operation is implicitly involved.

We use mainly:

_fetch_and_add to implement counters, statistics

_fetch_and_and/or to implement bit masks

_compare_and_swap to implement some carefully chosen list insertion/deletions

primitives (singly-linked lists)

6. Other SW issues on an SMP system

6.1 Scheduling and dispatching: affinity

The architecture being what it is, the natural tendency is to let the scheduler dispatch threads on processors from a single run-queue and according to priorities.

However, the presence of the large Level-2 caches creates some affinity between threads and processors because the working set of threads tends to accumulate in the caches [Tor92]. So if the thread is migrated to another CPU and therefore another pair of caches, a warm-up of the cache is unavoidable, with a burst of lateral misses (i.e. miss where data comes from other caches, as opposed to vertical misses where data comes from main memory). This temporarily high miss rate decreases the performance.

So we implemented an affinity scheduler with a simple algorithm to ensure that the pathlength increase in the scheduler code is minimal, so that the global outcome is positive. Each thread remembers its level of affinity with processors (actually, it is simply the last processor where the thread executed). When a processor becomes idle, the scheduler scans the runqueue and dispatches preferably a thread with affinity for it. The algorithm must however ensure fair access to CPU time. Several parameters (e.g. length and depth of the runqueue scan) were tuned to insure this.

6.2 Binding threads to processors

An API allows the programmer to explicitly bind a thread or a process to a specific processor, preventing the scheduler to migrate it when possible. Caution is recommended with this user-driven scheduling, because the scheduler generally does a better job in terms of global performance optimization. It should therefore be used only for special dedicated applications (e.g. real-time).

6.3 Managing interrupts

Masking against interrupts is done by writing some mask value into a HW register associated with the CPU. When the CPU is running at thread level, i.e with no interrupts masked, it writes a code in this register to distinguish the idle task from the normal processing. This register is used by HW to dispatch interrupts to the "least loaded processor", so all interrupts will go to one of the idle processors or, if none, to one of the processors not masked against interrupts.

As interrupt handling involves significant overhead, this "interrupt steering" improves response time. Maximum throughput is not much improved since, in that case, CPUs are seldom idle.

6.4 Clock management

Ideally, only one clock interrupt per system (as opposed to one clock interrupt per CPU) is needed on an SMP since almost all clock related actions are independent of the CPU. However, actions such as profiling a CPU i.e. determining which piece of code it is currently executing, need interruption of the CPU. So we ended up with a scheme where each CPU is interrupted at each clock tick but only one of them implements the system-wide time-related tasks.

6.5 Multi Processor communication (MPC)

SMP kernel implementations require some sort of basic inter-CPU communication. In AIX Version 4, it is called MPC.

The MPC allows a processor to send an interrupt to another processor and have it perform a specific action. We tried to maintain this layer as simple as possible. The MPC services allow:

- To register a service routine, which will be called by the target processor upon reception of an inter CPU interrupt.
- To trigger an inter CPU interrupt on a specific processor or on all the other processors (broadcast).

The MPC services are used for various functions such as funneling, timer management, dump and kernel debugger (to stop all the other processors).

7. Overall results

The results of all these technical choices can be summarized in scalability figures for various types of workloads. These figures being highly sensitive, you will have to refer to the marketing brochures of both companies to get them. Suffice it to say that from a technical standpoint we are very happy with the overall scalability.

8. How we did it

AIX 4.1 is the result of a joint development program between Bull (Grenoble, France) and IBM (Austin, TX). Bull brought its SMP experience in DPX/2 and DPS7000 and IBM its knowledge of AIX and the PowerPC architecture.

The architectural design was done by a joint team of SW architects. The work sharing was done by a joint team of technical managers. The component level design was done by the teams where each component responsibility had been assigned.

A common team in a single location did the locking of the kernel core because these parts interact so heavily that a single team was necessary. The team was composed of engineers from both companies.

The rest of the work in kernel, libraries and commands was separable and so could be dispatched between Austin and Grenoble with the help of a fast transatlantic communication link allowing us to work on the same source tree with a unique configuration management system in almost real time.

The whole project lasted all in all more than 2 years, from early Bull-IBM technical contacts to shipment. However, this involves the SMP HW development and the entirety of the AIX Version 4 development cycle. This latter decision must be drawn to the attention of those who may have heard of shorter development cycles with respect to MP-enabling a UNIX OS implementation. Instead of freezing the functionality of AIX 3.2 to make it support SMP HW and let AIX evolve functionally in parallel to meet market requirements on UP only, we commonly decided to proceed with a single development path. This by no doubt made the MP enablement more complex, but avoided the confusion and costs resulting from having 2 versions of AIX.

So it is difficult to size the SW SMP effort per se, but it lasted around 18 months (design to General Availability) and involved more than 100 people.

With a lot of good will on both sides, this cooperation worked and is still working incredibly well, probably due to our common technical culture which is a mix of UNIX background and mainframe experience.

9. References

This is not a complete bibliography on MP aspects of the UNIX kernel, because it would be too big.

- [Cam91] Lock Granularity Tuning Mechanisms in SVR4/MP, Mark D. Campbell, Russ Holt, John Slice, USENIX SEDMS II, March 1991, p. 221
- [Car93] Measuring Lock Performance in Multiprocessor Operating System Kernels Joseph P. CaraDonna, Noemi Paciorek, Craig E. Wills, USENIX SEDMS IV, Sept 1993, p. 37
- [IBM94] PowerPC and POWER2: Technical Aspects of the New IBM RISC System/6000, Prentice-Hall 1994
- [Kor89] Sema: a LINT-like tool for analyzing semaphore usage in a multithreaded UNIX kernel, Joe Korty, 1989 Winter USENIX

- [Pac91] Debugging Multiprocessor Operating System Kernels, N.Paciorek, S.LoVerso, A.Langerman, USENIX SEDMS II, March 1991, p. 185
- [Pow91] SunOS Multi-thread Architecture, M.L. Powell, S.R. Kleiman, S. Barton, D. Shah, D. Stein, M. Weeks, 1991 Winter USENIX, p. 65
- [Rit84] A Stream Input-Output System, D.Ritchie, AT&T Bell laboratories Technical Journal, Vol. 63 no. 8 (October 1984)
- [Ste93] WARLOCK A Static Data Race Analysis Tool, N.Sterling, 1993 Winter USENIX, p. 97
- [Tor92] Evaluating the benefits of cache affinity scheduling in shared memory multiprocessors, J.Torellas, A.Tucker, A.Gupta Technical report: CSL-TR-92-536 (Stanford Univ.) August 1992

10. Trademarks

Escala, PowerScale, BOS, DPS and DPX are trademarks of Bull S.A.

AIX, RS/6000 and PowerPC are registered trademarks of IBM

UNIX is a registered trademark licensed exclusively through X/Open

OSF/1 is a registered trademark of OSF

11. Author information

Jacques Talbot is responsible of SW Architecture in the Open Systems department at Bull Grenoble since 1989. He was previously in charge of the Bull BOS UNIX kernel development, and then project manager for the Bull DPX/2 200 UNIX platform. He graduated in 1972 from the Ecole Supérieure d'Electricité.

Address: 1, rue de Provence 38130 Echirolles FRANCE

Email: J.Talbot@frec.bull.fr

This paper has been written by Jacques Talbot from Bull. However, it is the result of the work of many people in both Bull and IBM. Special thanks to Jack O'Quin, John O'Quin, Jeff Peek from IBM and André Albot, Corrado Clementi, Jean-Jacques Guillemaud, Steve Hinde (*), Michel Sanchez from Bull.

(*) now with another company

A Flash-Memory Based File System

Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda Advanced Research laboratory, Hitachi, Ltd.

Abstract

A flash memory device driver that supports a conventional UNIX file system transparently was designed. To avoid the limitations due to flash memory's restricted number of write cycles and its inability to be overwritten, this driver writes data to the flash memory system sequentially as a Log-structured File System (LFS) does and uses a cleaner to collect valid data blocks and reclaim invalid ones by erasing corresponding flash memory the Measurements showed that the overhead of the cleaner has little effect on the performance of the prototype when utilization is low but that the effect becomes critical as the utilization gets higher, reducing the random write throughput from 222 Kbytes/s at 30% utilization to 40 Kbytes/s at 90% utilization. The performance of the prototype in the Andrew Benchmark test is roughly equivalent to that of the 4.4BSD Pageable Memory based File System (MFS).

1. Introduction

Flash memory, a nonvolatile memory IC (Integrated Circuit) that can hold data without power being supplied, is usually a ROM (Read Only Memory) but its content is electrically erasable and rewritable. The term "flash" is used to indicate that it is a whole chip or a block of contiguous data bytes (We call this block an *erase sector*). Many kinds of flash memory products [1][2] are available, and their characteristics are summarized in Table 1[†].

Because flash memory is five to ten times as expensive per megabyte as hard disk drive (HDD) memory, it is not likely to become the main mass storage device in computers. Its light weight, low energy consumption, and shock resistance, however,

Read Cycle	80 - 150 ns
Write Cycle	10 μs/byte
Erase Cycle Cycles limit Sector size	1 s/block 100 000 times 64 Kbytes
Power Consumption	30 - 50 mA in an active state 20 - 100 μA in a standby state
Price	10 - 30 \$/MByte

Table 1. Flash memory characteristics.

make it very attractive for mass storage in portable computers. In fact, flash memory in the form of an IC-card commonly replaces the HDD or is used for auxiliary storage in portable personal computers.

Flash memory has two other disadvantages limiting its use in computer systems. One is that its content cannot be overwritten: it must be erased before new data can be stored. Unfortunately, this erase operation usually takes about one second. The other disadvantage is that the number of erase operations for a memory cell is limited, and upper limits on the order of 100 000 erasures are not unusual. An advantage of flash memory, however, is that its read speed is much greater than that of a HDD. The performance of flash memory in read operations is, in fact, almost equivalent to that of conventional DRAM.

Our objective in the work described here was to explore the possibilities of using flash memory in file systems and to develop an experimental but practical flash memory based file system for UNIX. We used a log approach to ensure that new data was always written in a known location—so that the erase operation could be performed in advance.

2. Design and Implementation

We have designed and implemented a flash memory device driver that emulates a HDD and supports a standard UNIX file system transparently. We chose the device driver approach for its simplicity.

[†]There is another type of flash memory that has much smaller erase sectors. (See Section 4.)

Since flash memory can be accessed directly through a processor's memory bus, other approaches (such as tightly coupling a flash memory system and a buffer cache) might perform better by reducing memory-to-memory copy operations. Such an approach, however, would require a large number of kernel modification because flash memory's erase and write properties differ greatly from those of the main memory.

2.1 Overview

Our driver must record which regions contain invalid data and reclaim the invalid region by erasing those regions. Furthermore, since the number of erase operations in each region is limited, the driver must at least monitor the number in order to assure reliable operation. In some cases, wear-leveling should be provided.

Our driver maintains a sequential data structure similar to that of LFS [3][4]. It handles a write request by appending the requested data to the tail of the structure. To enable later retrieval of the data it maintains a translation table for translating between physical block number and flash memory address. Because the translation is made on the level of the physical block number, our driver can be viewed, from the file-system aspect, as an ordinal block device.

When write operations take place the flash memory system is fragmented into valid and invalid data blocks. To reclaim invalid data blocks, we use a cleaner that selects an erase sector, collects valid data blocks in the sector, copies their contents to the tail of the log, invalidates the copied blocks (and consequently makes the entire sector invalid), and issues an erase command to reclaim the sector. The functions of this cleaner are identical to those of LFS's cleaner. Our prototype does not implement wear-leveling, although it does maintain a log of each erased sector.

2.2 Flash Memory Capability

Early generations of flash memory products prohibit read or write operations while they are performing a write or an erase operation. That is, when such a flash memory chip is performing an erase operation on an erase sector, data can neither be read from nor written to other erase sectors until the erase operation completes. A naive file system implementation ignoring this prohibition would not be feasible in a multitasking environment because it would unpredictably block operations whenever the program wanted data from a flash memory chip that

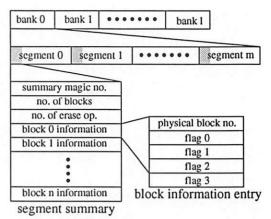


Figure 1. On-chip data structure.

was performing an erase operation triggered by another program. This problem can be avoided by temporarily caching in a buffer all valid data in the flash memory chip to be erased. This caching operation, however, could consume a significant amount of processor resources.

Recent flash memory products provide an erasesuspend capability that enables sectors that are not being erased to be read from during an erase operation. Some new products also support write operations during the erase suspension. Our driver assumes the underlying flash memory system to be capable of erase- suspended read operations.

Flash memory generally takes more time for a write operation than for a read operation. It provides a write bandwidth of about 100 Kbytes/s per flash memory chip, whereas a conventional SCSI HDD provides a peak write bandwidth 10 to 100 times higher. Some recent flash memory products incorporate page buffers for write operations. These buffers enable a processor to send block data to a flash memory chip faster. After sending the data, the processor issues a "Page Buffer Write" command to the chip and the chip performs write operations while the processor does other jobs.

Our driver assumes that the underlying flash memory system consists of some banks of memory that support concurrent write operations on each chip. This assumption reduces the need for an on-chip page buffer because the concurrent write operations can provide a higher transfer rate.

2.3 On Flash Memory Data Structure

Figure 1 depicts our driver's data structure built on an underlying flash memory system. The flash memory system is logically handled as a collection of banks. A bank corresponds to a set of flash memory chips and each set can perform erase or write operations independently. The banks are in turn divided into segments, each of which corresponds to an erase sector of the flash memory system. Each segment consists of a segment summary and an array of data blocks. The segment summary contains segment information and an array of block information entries. Segment information includes the number of blocks the segment contains and the number of times the segment has been erased.

2.4 Flag Update

Each block information entry contains flags and the physical block number to which this data block corresponds. The physical block number is provided to the driver by the file-system module when issuing a write data request. The flags are written sequentially so that the driver can record the change of the block status without erasing the segment.

The driver uses four flags to minimize the possibility of inconsistency and to make recovery easier. When a logical block is overwritten the driver invalidates the old block, allocates a new block, and writes new data to the newly allocated block. The driver updates the flags on the flash memory in the following order:

Step 1. mark the newly allocated block as allocated,

Step 2. write the block number and then write new data to the allocated block,

Step 3. mark the allocated block as pre-valid,

Step 4. mark the invalidated block as invalid, and

Step 5. mark the allocated block as valid.

The above steps guarantee that the flag values of the newly allocated and invalidated blocks never become the same under any circumstances. Therefore, even after a crash (e.g., a power failure) during any one of the above steps, the driver can choose one of the blocks that hold the fully written data. This method is of course not sufficient to maintain complete file system consistency, but it helps suppress unnecessary ambiguity at the device level.

2.5 Bank Management

To manage block allocation and cleaning, the driver maintains a bank list and a cleaning bank list. Figure 2 shows the relationship between these lists. The driver allocates a new data block from the active bank, so data write operations take place only on the active bank. When the free blocks in the active bank are exhausted, the driver selects from the bank list the bank that has the most free segments (i.e., free blocks) and makes it the new active bank.

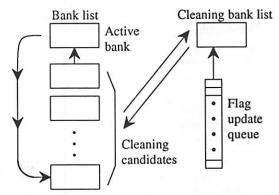


Figure 2. Bank list and cleaning bank list.

When a segment is selected to be cleaned, the bank containing that segment is moved to the cleaning bank list. The bank stays in the list until an erasure operation on the segment finishes. Because the bank is no longer on the bank list, it never becomes an active bank, and thus avoids being written during the erase operation.

The driver maintains the flag update queue to handle the flag update procedure, described in the previous section, on blocks in the bank of which segment is being erased. The driver avoids issuing a data write on a bank being cleaned by separating the bank list and the cleaning bank list. However, when a block is logically overwritten, an invalidated block might belong to that bank. In such a case, the driver postpones the flag update procedure steps 4 and 5, by entering the pair of the newly allocated and the invalidated blocks into the queue. All the pairs are processed when the erasure finishes. Note that even if the pairs are not processed due to a crash during the erasure, the driver can recover flag consistency because of the flag update order (Step 3 for each pair has been completed before the crash occurs.).

The queue should be able to hold the number of pairs that are expected to be entered during an erasure. For example, the current implementation can generate 500 pairs for 1 erasure [i.e., 500 blocks (250 KBytes) per second], and thus has 600 entries in the queue. Should the queue be exhausted, the driver will stop writing until the erasure is complete. We have not yet experienced this condition.

2.6 Translation Table

The translation table data structure contains all information needed to manage the translation of a block number to an address and to manage the erase log of each segment. During the system boot time, the driver scans the flash memory system and constructs this translation table and other structures from the on-chip segment summaries.

Figure 3 shows the relationship between the translation table and the block information entries. During the system boot time the driver scans all the segment summaries one by one. If it finds a valid block, it records a triplet (bank no., segment no., block no.) describing the block in a table entry indexed by the physical block number.

After the boot, the driver refers only to the translation table to access data blocks on the flash memory when a read operation is requested. The address of each block can be computed from the triplet. The driver translates a requested physical block number to the address of a corresponding flash memory data block and simply copies the contents of the data block to the upper layer.

When a write operation is requested, the driver checks whether it has already allocated a flash memory data block for a requested physical block. If it has, the allocated block is invalidated. The driver allocates a new flash memory data block, updates the translation table, and copies the requested data to the newly allocated block, while updating the flags.

2.7 Cleaner

The segment cleaning operation takes place during the allocation process when the number of available flash memory blocks for writing becomes low. This operation selects a segment to be cleaned, copies all valid data in the segment to another segment, and issues a block erase command for the selected segment. The cleaning process is the same as that of LFS except that it explicitly invokes the erase operation on the segment.

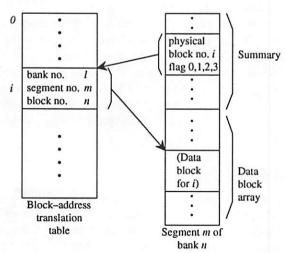


Figure 3. Relationship between the block-address translation table and a block information entry.

Command	Description			
FLIOCCWAIT	Wait until a segment is selected to be cleaned.			
FLIOCCCBLK	Copy 16 valid blocks of the selected segment. Return 0 if no more valid blocks exist in the segment.			
FLIOCCIERS	Start erasing the segment. Return 0 when the erasure is complete.			

Table 2.ioctl commands for the cleaner.

The cleaner is divided into three parts: policy, copying and erasing, and internal data maintenance. All jobs are executed in kernel-space, though copying and erasing are conducted by a daemon running in user-space.

As discussed in [4], implementing a cleaner as a user process makes the system flexible when changing or adding a cleaning policy or algorithm, or both. By communicating with the kernel through the use of system calls and the *ifile*, the BSD LFS cleaner does almost all jobs in user-space. Our driver, in contrast, does the cleaning jobs in kernel-space as Sprite LFS does. We use a daemon to make the copying process concurrent with other processes. We took this approach for its ease of implementation.

While data is being written, cleaning policy codes are executed when a block is invalidated. If cleaning policy conditions are satisfied for a segment, the driver adds it to the cleaning list and wakes up the cleaner daemon to start copying valid blocks. Upon awakening, the cleaner daemon invokes the *copy* command repeatedly until all valid blocks are copied to the clean segments. Then, it invokes the *erase* command and the driver starts erasing the segment by issuing an erase command of the flash memory. The copying is performed by codes within the driver. The cleaning daemon controls the start of the copying. It makes the copying concurrent with other processes.

We added three ioctl commands for the cleaner daemon (Table 2). The daemon first invokes FLIOCCWAIT and then waits (usually) until a segment to be cleaned is selected. As an application program writes or updates data in the file system, the device driver eventually encounters a segment that needs to be cleaned. The driver then wakes up the cleaner daemon and continues its execution. Eventually, the daemon starts running and invokes FLIOCCCBLK repeatedly until all the valid blocks are copied to a new segment. On finishing the copy operation, the daemon invokes FLIOCCIERS, which causes the driver to issue an erase command to the flash memory. The daemon invokes FLIOCCWAIT again and waits until another segment needs to be cleaned.

2.8 Cleaning Policy

For our driver, the cleaning policy concerns:

- · When the cleaner executes, and,
- · Which segments is to be cleaned.

The flash memory hardware permits multiple segments to be erased simultaneously as long as each segment belongs to the different bank. This simultaneous erasure provides a higher block-reclaim rate. For simplicity, however, the current implementation cleans one segment at a time. The cleaner never tries to enhance logical block locality during its copying activity. It simply collects and copies live data in a segment being cleaned to a free segment.

In order to select a segment to clean, the driver is equipped with two policies: "greedy" and "cost-benefit" [3] polices. The driver provides ioctl commands to choose the policy. The greedy policy selects the segment containing the least amount of valid data, and the cost-benefit policy chooses the most valuable segment according to the formula:

$$\frac{\text{benefit}}{\text{cost}} = \frac{age \times (1-u)}{2u},$$

where u is the utilization of the segment and age is the time since the most recent modification (i.e., the last block invalidation). The terms 2u and 1-u respectively represent the cost for copying (u to read valid blocks in the segment and u to write back them) and the free space reclaimed. Note that LFS uses 1+u for the copying cost because it reads the whole segment in order to read valid blocks in the segment for cleaning.

The cleaning threshold defines when the cleaner starts running. From the point of view of the load put upon the cleaner, the cleaning should be delayed as long as possible. The delay should produce more blocks to be invalidated and consequently reduce the number of valid blocks that must be copied during the cleaning activity. Delaying the cleaning activity too much, however, reduces the chances of the cleaning being done in parallel with other processes. This reduction may markedly slow the system.

Our driver uses a gradually falling threshold value that decreases as the number of free segments decreases. The curve A in Figure 4 shows the threshold of the current implementation. It shows that

- When enough (N) free segments are available, the cleaner is not executed,
- When the number of free segments becomes smaller than N and if there are some segments whose policy accounts are greater than T_h , cleaning starts with the segment that has the greatest policy accounts, and

 As the number of free segments becomes smaller, the threshold becomes lower so that more segments can be chosen with lower policy accounts.

For the greedy policy of the current implementation, N is 12 and T_h is 455 invalid blocks. That is, when the number of free segments becomes 12, segments that contain more than 455 invalid blocks are cleaned. For the cost-benefit policy, N is 12 and T_h is set to the value that is equivalent to being unmodified 30 days with one invalid block. For both the policies, segments having no valid blocks are always cleaned before other segments.

The threshold curve enables the driver to stop the cleaner as long as enough free segments are available and also to start the cleaner at a slow pace. For example, suppose the driver employs a policy such as "When the number of free segments becomes N_b , start the cleaner." (This policy is represented by the threshold curve B in Figure 4.) When the number of free segments became N_b the cleaner would start cleaning even if the most invalidated segment had only one invalid block. Furthermore, if the live data in the file system counted more than N_s - N_b segments (where N_s is a total number of segments), the cleaner would run every time a block was invalidated. This would result in a file system that was impractically slow and had an impractically short lifetime.

3. Performance Measurements and Discussion

Unlike a HDD-based file system, the prototype is free from seek latency and it is thu expected to show nearly the same performance for both sequential and random read operations. In fact, for reading 4Kbyte blocks from 12.6 Mbytes of data, the sequential and random throughputs of the driver are respectively 644 and 707 Kbytes/s. (For the same tasks, the through throughputs of MFS [7] are 647 and 702 Kbytes/s.)

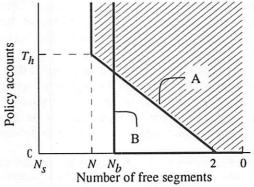


Figure 4. Cleaning threshold.

Flash Memory Syste	em				
Flash memory	Intel 28F008SA (1 Mbyte/chip)				
Banks	8				
Segments	64 (8 segments/bank)				
Segment size	256 Kbytes				
Data blocks	32256 (504 blocks/segment)				
Erase cycle	1 sec.				
bandwidth	252 Kbytes/sec.				
Write bandwidth	400 Kbytes/sec.				
Read bandwidth	4 Mbytes/sec.				
CPU and Main Men	nory System				
CPU	IDT R3081 (R3000 compatible)				
Cache	Instruction 16 Kbytes				
	Data 4 Kbytes				
Memory bandwidth					
Instruction read	20 Mbytes/sec.				
Data read	7 Mbytes/sec.				
Data write	5 Mbytes/sec.				

Table 3. Test-platform specifications.

The write performance of the prototype, on the other hand, is affected by the cleaning, as is the case with LFS

The benchmark platform consists of our hand-made computer running 4.4BSD UNIX with a 40MHz R3081 processor and 64 Mbytes of main memory. The size of the buffer cache is 6 Mbytes. Table 3 summarizes the platform specification[†].

3.1 Sequential Write Performance

The goal of our sequential write performance test was to measure the maximum throughput that can be expected under certain conditions. When a large amount of data is written sequentially, our driver invalidates blocks in each segment sequentially. The driver therefore needs no copying for cleaning a segment and the maximum write performance can be obtained.

Figure 5 shows the sequential write throughput as a function of cumulative data written, and Table 4 summarizes the results. The results were obtained by first writing a certain amount of data and then repeatedly overwriting that initial data. The curves show results based on different initial data: 4.2 Mbytes (30% of the file system capacity), 8.4 Mbytes (60%), and 12.6 Mbytes (90%). The greedy policy was used for cleaning.

The results obtained with the 90% initial data load were unexpected. Although the data were overwritten sequentially, many blocks were copied for cleaning. This copying was a result of the effect of the cleaning threshold described earlier. Since the live data counts

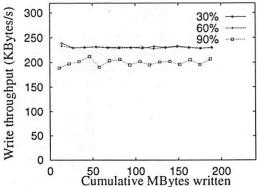


Figure 5. Sequential write performance.

Initial data	Average throughput (Kbytes/s)	Number Number of erased of copied segments blocks		Total data written (Mbytes)
30%	231	749	0	192
60%	230	766	0	192
90%	199	889	57 266	192

Table 4. Summary of sequential write performance.

more than 53 segments for the 90% data, the cleaning threshold was kept near 410 invalid blocks in a segment throughout the test. Consequently, the cleaner copied an average of 64 blocks per erasure and lowered the write throughput.

3.2 Random Write Performance

The random write performance test evaluated the worst case for our driver. When a randomly selected portion of a large amount of data is overwritten, all the segments are invalidated equally. If the invalidation takes place unevenly (e.g., sequentially), some segments are heavily invalidated, and thus can be cleaned with a small amount of copying. The even invalidation caused by the random update, however, results in there being less chance to clean segments that are particularly highly invalidated. Therefore, the cleaning cost approaches a constant value for all segments.

For our driver, the copying cost is expected to be a function of the ratio of used space to free space in the file system. As new data are written to the free segments, the used segments are invalidated evenly. The free segments are eventually exhausted and the cleaner starts cleaning. Consequently, the ratio of valid to invalid blocks of each segment becomes that of the ratio of used to free space of the file system.

Figure 6 and Table 5 show the results of the random write test. These results were obtained by writing a 4Kbyte data block to a randomly selected position of various amounts of initial data: again, 4.2,

[†] The actual hardware had 128 segments (16 segments/block), but in the work reported here we used only half the segments of each bank.

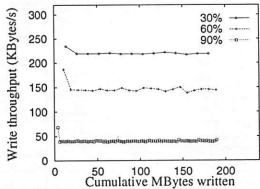


Figure 6. Random write performance.

Initial data	Average throughput (Kbytes/s)	Number of erased segments	Number of copied blocks	Total data written (Mbytes)
30%	222	801	26 383	192
60%	147	1066	155 856	192
90%	40	2634	938 294	192

Table 5. Summary of random write performance.

8.4, and 12.6 Mbytes. And again the greedy policy was used for cleaning.

3.3 Hot-and-Cold Write Performance

This test evaluated the performance cases where write accesses exhibited certain amounts of the locality of reference. In such cases, we can expect the cost-benefit policy to force the segments into a bimodal distribution where most of the segments are nearly full of valid blocks and a few are nearly full of invalid blocks [3]. Consequently, the cost-benefit policy would result in a low copying overhead. We will see that our first results did not match our expectations; we will then analyze why this anomaly occurred and what measures we took to address it.

Figure 7 and Table 6 show the results of the test. 640-116 means that 60% of the write accesses go to one-eighth of the initial data and other 40% go to another one-eighth. The other three-fourths of the data are left unmodified. With this distribution we intend to simulate meta data write activity on an actual file system. The ratio of writes is based on the results reported in [5], which found that 67-78% of writes are to meta data blocks. In all the tests we conducted, an actual write position in a selected portion of the initial data was decided randomly, the size of the initial data was 12.6 Mbytes (90%), and all writes were done in 4Kbyte units.

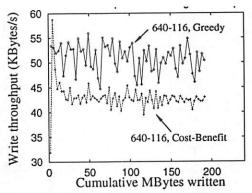


Figure 7. Hot-and-cold write performance.

Test	Cleaning policy	Average throughput (Kbytes/s)	Nomuber of erased segments	Nomuber of copied blocks			
Line In	Greedy	51	2617	925 193			
640-116	Cost- Benefit	43	3085	1 161 090			

Initial data: 90% (12.6 Mbytes), Total data written: 192 Mbytes

Table 6. Summary of hot-and-cold write performance.

3.4 Separate Segment for Cleaning

The initial results obtained in the hot-and-cold write test were far worse than we had expected. The write throughput of the 640-116 test was nearly the same as that of the random test using the greedy policy. Furthermore, the greedy policy worked better than the cost-benefit policy for the 640-116 test.

Figure 8 shows the distribution of segment utilization after the 640-116 test. In the figure, we can observe a weak bimodal distribution of segment utilization. Since the 60% of the data was left unmodified, more fully valid segments should be present.

We traced the blocks that the cost-benefit policy once judged as cold in the test, and Figure 9 shows the distribution of the cold and the not-cold blocks in the segments after executing the 640-116 test. The data in

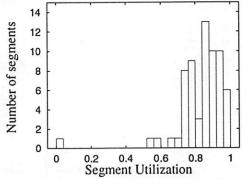


Figure 8. Segment utilization distribution after the 640-116 test.

Initial data	Separate segment	Average throughput (Kbytes/s)	Number of erased segments	Number of copied blocks
200	No	241	742	0
30%	Yes	. 239	744	0
60%	No	197	888	63 627
	Yes	198	832	33 997
70%	No	135	1195	214 894
706	Yes	143	883	57 205
80%	No	81	1855	544 027
	Yes	127	1089	157 922
90%	No	43	3085	1 161 090
	Yes	60	2218	723 582

Total data written: 192 Mbytes

Table 7. Summary of 640-116 tests using the separate cleaning segment.

this figure was obtained by marking a block as "cold" when the segment to which the block belongs was chosen to be cleaned and its utilization was less than the average utilization in the file system. We can see that some segments contain both cold and not-cold blocks. Furthermore, the number of cold blocks is much smaller than expected: since three-fourths of the 12.6 Mbytes of initial data were left unmodified, we would expect, in the best case, about 19 000 cold blocks (i.e., about 38 cold segments). In the test, however, the actual number of cold blocks was 2579.

The reason we determined for the above results is that the driver uses one segment for both the data writes and the cleaning operations; the valuable, potentially cold blocks are mixed with data being written to the segment. The number of cold blocks therefore does not increase over time.

To address this problem, we modified the driver so that the driver uses two segments: one for cleaning cold segments and one for writing the data and cleaning the not-cold segments. Table 7 summarizes the results of 640-116 tests on both the modified and the original drivers. The effect of the separate cleaning segment becomes notable as the initial utilization grows, and the write throughput was improved more than 40% for the 90% initial data. Figure 10 shows the

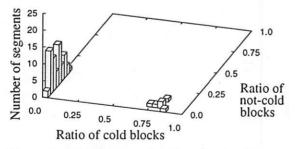


Figure 9. Distribution of cold and not-cold blocks after the 640-116 test.

		MFS	Proto	otype
		MFS	52-56%	92-96%
Average	Phase 1	1.3	2.0	2.9
elapsed	Phase 2	8.0	9.5	11.5
time	Phase 3	13.1	13.5	13.5
for	Phase 4	16.9	16.9	17.1
each	Phase 5	80.6	81.8	84.0
run	Total	119.9	123.7	129.0
Number blocks fo	of written or data		251 818	233 702
Number blocks	of copied		75 227	255 020
Number segments	of erased		578	903

Table 8. Andrew Benchmark results.

distribution of cold blocks after the 640-116 test using the modified driver. Many cold segments are observed.

3.5 Andrew Benchmark

Table 8 lists the results of the Andrew benchmark [6] for MFS and for our prototype. The results were obtained by repeating the benchmark run 60 times. The output data files and directories of each run were stored in a directory, and to limit the file system usage the oldest directory was removed before each run after 14 contiguous runs for the 52-56% test, after 24 for the 92-96% test, and after 9 for the MFS test. Note that, as pointed out in [4], phases 3 and 4 performed no I/O because all the data access were cached by the higher-level buffer and the *inode* caches.

The benchmark consists of many read operations and leaves a total of only about 560 Kbytes of data for each run. As a result, there are many chances to clean segments without disturbing data write operations. Therefore, our prototype shows performance nearly equivalent to that of MFS. We expect that similar access patterns often appear in a personal computing environment. Note that the cleaner erased 903

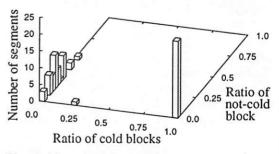


Figure 10. Distribution of cold and not-cold blocks after the 640-116 test using the separate cleaning segment.

segments for the 92-96% test; under the same load (15 segments in 2 minutes), our prototype will survive about 850 000 minutes (590 days).

4. Related Work

Logging has been widely used in certain kinds of devices; in particular, in Write-Once Read-Many (WORM) optical disk drives. WORM media are especially suitable for logging because of their append-only writing. OFC [8] is a WORM-based file system that supports a standard UNIX file system transparently. Although its data structures differ from those of our prototype, our prototype's block-address translation scheme is very similar to that of OFC. OFC is self-contained in that it stores all data structures on a WORM medium and needs no read-write medium such as a HDD. To get around the large memory requirement, it manages its own cache to provide efficient access to the structure. Our prototype, however, needs improvement with regard to its memory requirement (about 260 Kbytes for a 16-Mbyte flash memory system).

LFS [3] uses the logging concept for HDDs. Our prototype is similar to LFS in many aspects, such as segment, segment summary, and segment cleaner, but LFS does not use block-address translation. LFS incorporates the FFS index structure into the log so that data retrieval can be made in the same fashion as in the FFS. That is, each *inode* contains pointers that directly point to data blocks. Our prototype, on the other hand, keeps a log of physical block modification.

LFS gathers as many data blocks as possible to be written in order to maximize the throughput of write operations by minimizing seek operations. Since flash memory is free from seek penalty, maximizing the write size does not necessarily improve performance.

The paper on BSD-LFS [4] reports that the effect of the cleaner is significant when data blocks are updated randomly. Under these conditions, each segment tends to contain fewer invalid data blocks and the cleaner's copying overhead accounts for more than 60% of the total writes. With our prototype, this overhead accounts for about 70% on the 90%-utilized file system.

Since flash memory offers a limited number of write/erase cycles on its memory cell, our driver requires the block translation mechanism. Logical Disk (LD) [9] uses the same technique to make a disk-based file system log-structured transparently. Although the underlying storage media and goals are different, both the driver and LD function similarly. LD does, though, provides one unique abstract interface called *block lists*. The block lists enable a file

system module to specify logically related blocks such as an *inode* and its indirect blocks. Such an interface might be useful for our driver by enabling it to cluster hot and cold data.

Douglis et al. [10] have examined three devices from the viewpoint of mobile computing: a HDD, a flash disk, and a flash memory. Their simulation results show that the flash memory can use 90% less energy than a disk-based file system and respond up to two orders of magnitude faster for read but up to an order of magnitude slower for write. They also found that, at 90% utilization or above, a flash memory erasure unit that is much larger than the file system block size will result in unnecessary copying for cleaning and will degrade performance.

flash-memory-based storage The eNVy [11] tries to provide high performance in a transaction-type application area. It consists of a large amount of flash memory, a small amount of batterybacked SRAM for write buffering, a large-bandwidth parallel data path between them, and a controller for page mapping and cleaning. In addition to the hardware support, it uses a combination of two cleaning policies, FIFO and locality gathering, in order to minimize the cleaning costs for both uniform and hot-and-cold access distribution. Simulation results show that at a utilization of 80% it can handle 30 000 transactions per second while spending 30% processing time for cleaning.

Microsoft Flash File System (MFFS) [2] provides MS-DOS-compatible file system functionality with a flash memory card. It uses data regions of variable size rather than data blocks of fixed length. Files in MFFS are chained together by using address pointers located within the directory and file entries. Douglis et al. [10] observed that MFFS write throughput decreased significantly with more cumulative data and with more storage consumed.

SunDisk manufactures a flash disk card that has a small erasure unit, 576 bytes [12]. Each unit takes less time to be erased than does Intel's 16Mbit flash memory. The size enables the card to replace a HDD directly. The driver of the card erases data blocks before writing new data into them. Although this erase operation reduces the effective write performance, the flash disk card shows stable performance under high utilization because there is no need to copy live data [10]. In a UNIX environment with FFS, simply replacing the HDD with the flash disk would result in unexpected short life because FFS meta data such as inodes are located at fixed blocks and are updated more often than user data blocks. The flash disk card might perform well in the UNIX environment if a proper wear-leveling mechanism were provided.

5. Conclusion

Our prototype shows that it is possible to implement a flash-memory-based file system for UNIX. The benchmark results shows that the proposed system avoids many of the problems expected to result from flash memory's overwrite incapability.

The device driver approach makes it easy to implement this prototype system by using the existing FFS module. But because the FFS is designed for use with HDD storage, this prototype needs to use a portion of the underlying flash memory to hold data structures tuned for a HDD. Furthermore, the separation of the device driver from the file system module makes the prototype system management difficult and inefficient. For example, there is no way for the driver to know whether or not a block is actually invalid until the FFS module requests a write on the block—even if the file for which the block was allocated had been removed 15 minutes before. A file system module should therefore be dedicated to flash memory.

Acknowledgments

We thank the USENIX anonymous referees for their comments, Douglas Orr for valuable suggestions and comments on the drafts, Fred Douglis for making his draft available to us, and Satyanarayanan-san and the Information Technology Center, Carnegie-Mellon University for providing us the Andrew Benchmark.

Microsoft and MS-DOS are registered trademarks of Microsoft Corporation.

References

- Advanced Micro Devices, Inc., "Am29F040 Datasheet", 1993.
- [2] Flash Memory, Intel Corporation, 1994.
- [3] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System", ACM Transactions on Computer Systems, Vol. 10, No. 1, 1992.
- [4] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin, "An Implementation of a Log-Structured File System for UNIX", Proc. '93 Winter USENIX, 1993.
- [5] C. Ruemmler and J. Wilkes, "UNIX disk access patterns", Proc. '93 Winter USENIX, 1993.
- [6] J. H. Howard, et al., "Scale and Performance in a Distributed File System", ACM Transactions on Computer Systems, Vol. 6, No. 1, 1988.
- [7] M. K. McKusick, M. J. Karels, and K. Bostic, "A Pageable Memory Based Filesystem", Proc. '90 Summer USENIX, 1990.

- [8] T. Laskodi, B. Eifrig, and J. Gait, "A UNIX File System for a Write-Once Optical Disk", Proc. '88 Summer USENIX, 1988.
- [9] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh, "Logical Disk: A Simple New Approach to Improving File System Performance", Technical Report MIT/LCS/TR-566, Massachusetts Institute of Technology, 1993.
- [10] F. Douglis, R. Cáceres, F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber, "Storage Alternatives for Mobile Computers", Proc. 1st Symposium on Operating Systems Design and Implementation, 1994.
- [11] M. Wu and W. Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Storage System", Proc. 6th International Conference on Architectural Support for Programming Languages and Operating Systems, 1994.
- [12] "Operating system now has flash EEPROM management software for external storage devices" (in Japanese), Nikkei Electronics, No. 605, 1994.

Author Information

Atsuo Kawaguchi is a research scientist at the Advanced Research Laboratory, Hitachi, Ltd. His interests include file systems, memory management system, and microprocessor design. He received B.S., M.S., and Ph.D. degrees from Osaka University. He can be reached at atsuo@harl.hitachi.co.jp.

Shingo Nishioka is a research scientist at the Advanced Research Laboratory, Hitachi, Ltd. His interests include programming languages and operating systems. He received B.S., M.S., and Ph.D. degrees from Osaka University. He can be reached at nis@harl.hitachi.co.jp.

Hiroshi Motoda has been with Hitachi since 1967 and is currently a senior chief research scientist at the Advanced Research Laboratory and heads the AI group. His current research includes machine learning. knowledge acquisition, visual reasoning, information filtering, intelligent user interfaces, and AI-oriented computer architectures. He received his B.S., M.S., and Ph.D. degrees from the University of Tokyo. He was on the board of trustees of the Japan Society of Software Science and Technology and of the Japanese Society for Artificial Intelligence, and he was on the editorial board of Knowledge Acquisition and IEEE Expert. He can be reached motoda@harl.hitachi.co.jp.

All the authors can be reached at Advanced Research Laboratory, Hitachi, Ltd. Hatoyama, Saitama, 350-03 Japan.

TRON:

Process-Specific File Protection for the UNIX Operating System

Andrew Berman, Virgil Bourassa and Erik Selberg

Department of Computer Science and Engineering

University of Washington

Seattle, WA 98195

Abstract

The file protection mechanism provided in UNIX is insufficient for current computing environments. While the UNIX file protection system attempts to protect users from attacks by other users, it does not directly address the agents of destruction—executing processes. As computing environments become more interconnected and interdependent, there is increasing pressure and opportunity for users to acquire and test non–secure, and possibly malicious, software.

We introduce TRON, a process-level discretionary access control system for UNIX. TRON allows users to specify capabilities for a process' access to individual files, directories, and directory trees. These capabilities are enforced by system call wrappers compiled into the operating system kernel. No privileged system calls, special files, system administrator intervention, or changes to the file system are required. Existing UNIX programs can be run without recompilation under TRON-enhanced UNIX. Thus, TRON improves UNIX security while maintaining current standards of flexibility and openness.

1. Introduction

This paper describes the design and implementation of $TRON^1$, a process–specific file protection mechanism for the $UNIX^2$ operating system. TRON

puts flexible, easy-to-use discretionary access control in the hands of the user, giving him the power to monitor and prevent "Trojan Horse" attacks and other undesirable file accesses. No special accounts, magic files, or actions by the system administrator are required.

TRON works in conjunction with the existing UNIX file protection mechanism. No recompilation of binary executables is required, and processes not protected by TRON incur no significant performance penalty. TRON facilitates protected cross—domain procedure calls by allowing client processes to temporarily grant their rights to server processes. TRON also accommodates various policies for handling access violations.

TRON allows users to express their intended process accesses in a natural, succinct fashion. Rights can be specified for files, directories and directory trees, to correspond to the structure of the UNIX file system. Invocation of access restrictions is separated from invocation of processes, providing reuse and flexibility.

2. Motivation

The user-oriented, access control list file protection mechanism provided in UNIX attempts to protect user's files from other users. But this is an incomplete view of file security. Files are not affected by users directly, but indirectly through the execution of programs.

As a result, UNIX files are susceptible to attack by the user on himself. Once a user logs in, his processes have all rights afforded to the user, and thus incorrect or malicious programs (e.g. Trojan Horses, computer virus-

Named after the heroic fictional protection program from the 1982 Walt Disney movie, TRON.
 UNIX is a registered trademark of X/Open Company Limited.

es, etc.[5]) as well as unintentional mistakes (e.g., typing "rm -fr *" in the wrong directory) executed by the user can perform actions the user did not intend nor foresee.

This susceptibility is exacerbated by the necessary and common method of granting temporary privileges in UNIX, namely, suid (set user id) and sgid (set group id). These commands allow a program's user to temporarily gain the rights of the program's owner. Such privileges have successfully been exploited to gain undesired access to user accounts and files by other users[12].

A common security technique to enhance UNIX file protection for programs such as ftp is to use the privileged command, chroot, to "change the root," causing the visible directory hierarchy to be replaced by a safe subdirectory. For ftp, this makes utilities such as 1s unreachable, so copies must be maintained in the safe subdirectory. An alternative requiring less maintenance would be to use the existing file structure while restricting the rights given the ftp daemon process—namely, full rights to the safe subdirectory combined with read and execute rights for necessary utilities. The utilities, when executed, would likewise require limited rights.

Although an individual user may have access rights to a large variety of files, restricting the rights granted to individual processes to those believed to be germaine can prevent abuse. We wish to provide UNIX users with a service that limits file access on a per–process basis, while still allowing the temporary extension of rights as necessary.

3. Design

TRON provides process–specific file protection to complement the existing UNIX user–specific file protection mechanism. TRON provides a means of executing processes within protected *domains*, in which the process has a restricted set of rights to files and directories, specified by a set of *capabilities*. Enforcement of rights is performed in the kernel to ensure security.

When a user first logs on, he has his full, normal UNIX file permissions by default¹. At any time, the user may

elect to execute processes in a more restrictive TRON domain. As a complementary protection mechanism, TRON does not allow a process to perform an action that would violate normal UNIX file permissions. Conversely, even processes with super–user privileges are restricted when executing in a protected TRON domain.

3.1. Capabilities

For protection, an agent requires access rights to a resource. A list of access rights associated with a resource is called an *access control list*[17]. Alternatively, a list of access rights associated with an agent is called a *capability list*[6]. The design of TRON is patterned after capability—list systems in that the access rights are associated with the agents of access, processes.

Unfortunately, UNIX does not lend itself to a traditional capability-based approach[15], in which each resource is identified uniquely. In UNIX, files and directories can have many names because of directory links and mounted file systems. For this reason, we have chosen to make TRON a "character string" protection service, meaning TRON capabilities are expressed as rights associated with a character string representing an object. An object may have several such character strings naming it, but TRON is unaware of the connection between them.

The character string representing a file or directory object is its *canonical* pathname. The canonical pathname is the character string representing the absolute, rooted path of the file or directory name. When not specified directly in canonical form, the pathname of a file or directory is determined relative to the current working directory.

TRON currently understands two types of objects—files and directories. The rights associated with an object are dependent upon the object's type, and have meaning only in the context of accesses related to that type. This generality may allow the concept of TRON

^{1.} Although system administrators are free to configure either or both of the login process and initial console shell to execute within restricted TRON domains as well.

capabilities to be extended to resources beyond files and directories, e.g., sockets.

Rights that can be specified for file access include: read, write (does not imply the right to delete the file), execute, delete, and modify UNIX permissions (e.g., with chmod)¹. To aid in specifying rights for several files at once, directory capabilities perform dual duty—they specify rights for a directory and the files contained within it. In addition to the file–specific rights, which apply to all files within the directory, directory rights include: create new files in the directory and create links to files in other directories. Link also requires full rights to the file being linked to. A final directory right, subtree, extends the rights to the directory's subdirectory tree. The subtree right conveys the other directory rights to all objects with canonicalized names beginning with the name of the directory.

The determination of whether an object is a file or a directory is made automatically at the time of creating the capability. We require that the named object exist at this time.

3.2. Domains

TRON protection domains provide a protection environment for every process. A TRON domain consists of a set of processes, a set of capabilities, and a violation handler. Each process executes in exactly one TRON domain. A normal fork operation causes a newly created process to inherit its parent's TRON domain. A modification of the fork operation, tron_fork, creates a new TRON domain containing a subset of the parent's TRON domain capabilities, in which to run the new process.

We initially considered providing more flexible TRON domain creation by using a separate call, <code>tron_restrict_domain</code>, to be used immediately following the <code>fork</code> command to create and enter a restricted domain, but concluded that combining the operations into the single <code>tron_fork</code> call provided overriding benefits. Because <code>tron_fork</code> first verifies that the newly created TRON domain will contain a strict subset of rights prior to creating the child pro-

cess, failure of the domain creation does not result in a new child process with more rights than the parent process intended, nor does the parent process have to rely on this child process for notification of the failure. Additionally, combining the domain creation with the process creation ensures that each process exists within a single, invariant TRON domain throughout its lifetime, facilitating both simple garbage collection of TRON domains as well as the ability to loan capabilities to other processes (see 3.4 below).

We likewise considered another alternative, tron_create_domain, to be called by the parent process prior to performing the fork. However, conveying the identity of the newly created TRON domain to the fork operation requires either a change to the existing interface, breaking all existing code making use of fork, or introducing a new global state variable to handle what is essentially a special case, adding to the process state clutter. In addition, a simple reference counting method for automatic garbage collection of TRON domains does not suffice for this alternative. Thus, we decided that the tron_fork functionality was the most appropriate for our design.

A special TRON domain, called the *full* TRON domain, gives full rights to all files and directories, effectively limiting access to the user's existing UNIX file permissions. This is the TRON domain of the first processes entered into the process table, namely init, pager, swapper, and idleproc. By default, progeny of these processes continue to inherit the full TRON domain until a tron_fork is performed. Thus, until activated, TRON does nothing beyond the normal functioning of the UNIX file protection mechanism. Therefore, all processes executed up to and including the first process performing a tron_fork are run with the full set of default UNIX rights.

3.3. Enforcement

TRON enforces domain–specific access rights from within the UNIX kernel. Placing TRON access enforcement within the kernel both ensures the security of the enforcement and obviates the need for recompiling existing executables. TRON access enforcement is localized to the *syscall* table, from which all system calls are dispatched upon entry into

^{1.} Our set of rights is inspired by those of the Andrew File System[11].

the kernel. TRON wrappers are placed around all pertinent system call operations (e.g. open, chmod, etc.). The syscall table is modified to vector control to the appropriate TRON wrapper for each system call. These wrappers perform the general algorithm given in Figure 1.

The enforcement algorithm directly invokes the desired kernel service for processes in full TRON domains. Limited TRON domains are examined for the necessary access rights to the desired service routine, invoking the service if found. Otherwise, a violation handler is called instead of the desired service routine. For processes not employing the TRON protection mechanism (i.e., in the full TRON domain), the performance impact of the additional if—test and procedure call is not significant relative to the cost of the kernel trap itself.

Note that it is possible for a file or directory to be interpreted with one canonicalized pathname at the time a capability is created, and a different canonicalized pathname at the time of enforcement, if either of both of the specifications are relative to the current working directory (due to links, mount points, etc.). When the names don't agree, the access attempt fails. This is a conservative solution to the name ambiguity problem inherent in the UNIX file system.

We envision the violation handlers to take on varied forms depending on the requirements of the computing site. Potential handler actions include soft responses, such as setting the offending process' global errno value to EACCES, or hard responses, such as killing the offending processes with a specific exit value. Logging and user intervention are other actions that may be taken. Because violation handlers are invoked from within the kernel, they must be built into the kernel itself, with policies established for their interaction with user-level processes.

3.4. Granting Capabilities

Our mechanism for temporarily granting access to another process consists of two system calls executed by the grantor of the capabilities, tron_grant and tron_revoke. This mechanism allows protected cross-domain procedure calls that are transparent to the serving process. The tron_grant call from a client process instructs the kernel to extend a given server process' TRON domain to include a subset of the client's capabilities. The extended capabilities are marked as revocable by the granting process. Once received, the server process, as well as any other process within the server process' TRON domain, has the extended access. In no case does the granting of capabilities provide file access to the server beyond the existing UNIX permission system. The extended TRON domain then allows servers to fork off copies of themselves to perform the desired service and still retain the necessary capabilities. By judicious creation of the server's TRON domain, it's straightforward to ensure that unintended processes do not share in the granted rights.

An additional system call, tron_get_cap_list, is available for determining the current domain capabilities. Processes are free to grant any or all of their

```
tron_foo_wrapper( foo_arguments )
{
   if ( tron_domain[ process ] ≠ FULL_TRON_DOMAIN ) {
      determine required_rights for foo( foo_arguments );
      if ( required_rights ⊄ capabilities[ tron_domain[ process ]] ) {
      invoke violation_handler[ tron_domain[ process ]];
        return from kernel;
      }
   }
   invoke foo( foo_arguments );
   return from kernel;
}
```

Figure 1. General System Call Wrapper Algorithm.

capabilities to other processes. Multiple capabilities for the same object granted to the same process are each individually added to the receiving process' TRON domain—any one of these providing the necessary rights suffices for access to the object.

The tron_revoke call removes all capabilities from a TRON domain that are marked as revocable by the calling process. This function serves more as a cleanup mechanism than as a security mechanism. We take the stance that the granting of a capability to another agent implies trusting that agent to be responsible with it. We use the tron_revoke call merely to keep servers' TRON domains from accumulating an indefinitely large number of capabilities.

The tron_revoke call is not transitive, e.g., if process A grants capability α to process B, which subsequently grants α to process C, then process A revoking its granted rights from process B does not imply revocation of α from process C. It is incumbent upon each direct grantor to explicitly revoke their granted rights prior to terminating. Thus, using tron_grant without a corresponding tron_revoke provides a capability hand—off mechanism.

A user-level command, <code>tron_loan</code>, provides for transparent capability granting and subsequent revocation on behalf of existing TRON-unaware client executables. The <code>tron_loan</code> command is executed from within an existing TRON domain. It first grants a specified subset of the TRON domain capabilities to an executing server process, then forks and executes the desired client program within the original domain. Upon completion of the client process, <code>tron_loan</code> revokes the granted capabilities from the server process. In other words, the <code>tron_loan</code> command acts as a proxy for a new client process to temporarily loan needed rights to a server process.

Although the tron_loan mechanism has some shortcomings for TRON-unaware executables, such

as for servers acting as clients of other servers, it does not preclude the transparent use of TRON in the general case. Rather the tron_loan command provides an extension of the normal TRON protection scheme for common cases by allowing server processes to maintain a minimal set of capabilities, while acquiring others as needed. We hope that with experience more general mechanisms will come to light.

4. Implementation

We have successfully implemented a prototype of the TRON service in ULTRIX V4.2A, running on a DECstation 5000/200¹. The impact on the existing system call code path is minimal when the protection is not in force and reasonably inexpensive when it is in force (on the order of the number of capabilities in the process' TRON domain). Modification of the existing ULTRIX source code is limited to small changes to a handful of files. Less than 2000 lines of new code are introduced into the kernel, while roughly 800 lines of code are introduced at the user level in the form of library calls and shell commands.

TRON functionality extends from user-level commands to internal kernel procedures. User commands consist of tron and tron_loan; system calls consist of tron_fork, tron_grant, tron_revoke, and tron_get_cap_list; and internal kernel procedures consist of the various TRON wrappers, tron_canonical, tron_verify, and tron_soft_violation_handler.

4.1. User Commands

The user commands introduced for TRON are summarized in Figure 2.

The tron command creates a TRON domain in which the given command is executed with the rights speci-

Figure 2. TRON User Command Summary

ULTRIX and DECstation are trademarks of Digital Equipment Corporation.

[%] tron [-p <rights_spec> [<file>|<dir>]...]... [-c <command_line>]

[%] tron_loan <pid> [-p <rights_spec> [<file>|<dir>]...]... [-c <command_line>]

[%] alias stdtron 'tron -p rx \$path ~ / -p rs /etc \!*'

fied. If no <command_line> is given, TRON invokes a shell (using the SHELL environment variable or /bin/sh if not found) in the new TRON domain. Note that the TRON domain specified must contain the rights to read and execute the given command. File and directory names are put into canonical form, verified to exist, and typed as a file or directory automatically. A <rights_spec> applies to all file and directory names encountered before the next -p or -c flag. The <rights_spec> is any combination of the following letters, which convey the associated rights:

r: read m: modify UNIX permissions

w: write c: create
x: execute 1: link
d: delete s: subtree

In practice, we have found the (c-shell) alias given in Figure 2, stdtron, to be helpful for common uses of tron. It conveys, at a minimum, read and execute rights to files on the search path, home, and root directories, and read rights to the /etc subtree.

The tron_loan command does not create a new TRON domain, but rather grants a subset of capabilities of the current TRON domain to the <pid> process' TRON domain, then forks the specified command within the current TRON domain. The capabilities are verified to be a subset of the current capabilities and, when granted, are marked as revocable by the tron_loan process. Like tron, when no <command_line> is given, tron_loan invokes a shell process. When the command or shell terminates, the capabilities are revoked from the <pid> process' TRON domain.

4.2. System Calls

The additional system calls introduced for TRON are summarized in Figure 3.

The tron_fork system call verifies that the capabil-

ities presented are a subset of the parent's TRON domain capabilities. If so, it creates a new TRON domain containing these capabilities and forks a new process within the new TRON domain. If the capabilities are incorrect, an error code is returned to the parent process and no fork is performed.

Like tron_fork, the tron_grant system call verifies that the capabilities presented are a subset of the process' TRON domain capabilities. If so, it appends the capabilities to the TRON domain containing the pid process, marking them as revocable by the current process.

The tron_revoke system call removes all capabilities marked as revocable by the current process from the TRON domain containing the pid process. If the pid process and the current process are identical no action is taken.

The tron_get_cap_list system call returns up to buf_size of the process' current capabilities to its cap_list_buffer, and returns the total number of capabilities in the TRON domain in num_caps.

4.3. Kernel Modifications

The modifications to the original kernel source are simple and straightforward. A TRON domain index is added to the process structure¹, fork is modified to copy this value, and exit is modified to clean up the TRON domain memory when no longer in use. Some initialization code is added to init_main.c. Finally, as alluded to earlier, syscall.h, syscalls.c, and init_sysent.c are modified to vector the appropriate system calls to the correspond-

```
tron_fork( cap_list, num_caps );
tron_grant( pid, cap_list, num_caps );
tron_revoke( pid );
tron_get_cap_list( cap_list_buffer, buf_size, num_caps );
```

Figure 3. TRON System Call Summary

^{1.} The addition of a field to the process structure was done for convenience in the prototype. A separate array of TRON domain indices to shadow the process table would allow existing system programs that depend on proc.h to run without recompilation.

Figure 4. TRON Kernel Function Summary

ing TRON wrappers.

The TRON wrapper procedures are straightforward applications of the algorithm described in section 3.3. In addition to the TRON wrappers, the support functions summarized in Figure 4 have also been added to the kernel.

The tron_canonical procedure is used to normalize file or directory names passed as arguments to system calls. It returns the canonical pathname of the given filename. The current working directory determination exactly mirrors the effective functionality of the user command, pwd. Local and remote mount points, hard and soft links, etc., are handled as they would be by the individual system calls made by pwd.

The tron_verify procedure is used to determine if the required rights are permitted by some capability in the current process' TRON domain. It searches through capabilities of the current process' TRON domain for any that convey the required_rights (a bit field with the corresponding bits for the required rights set to one). This includes capabilities for parent directories with the subtree right.

The tron_soft_violation_handler procedure is our prototype access violation handler. This procedure sets the current process' global errno value to error_status (generally set to EACCES) and

the system call return value to return_value (typically -1). It does no reporting of the access violation.

5. Examples

TRON usage examples are given in Figure 5. Example A runs a normal shell, but prevents deletion or writing of any files.

Example B runs a shell that allows deletion and writing in your current directory and /tmp. Note that rights are given to the current directory at the time the command is given, so the rights don't "float" if the current directory changes.

Example C allows use of emacs in the background to edit files only in the current directory (assuming all emacs support files are in the /usr/local/emacs subdirectory).

Example D creates a TRON domain to hold the finger daemon, permitting normal execution of its functions. This would have prevented the Internet Worm from taking advantage of the bug in fingerd to infiltrate systems[18]. Similar TRON domains can be created for each daemon in /etc/inetd.conf.

If users wished to have their .plan and .project files accessible to the finger daemon only when they are logged in and are alerted of their being "fingered," this

```
A) % tron -p rxmcls /
```

Figure 5. TRON Usage Examples

B) % tron -p rxmcls / -p wd . /tmp

c) % stdtron -p rwcds /usr/local/emacs -p rw . -c emacs myfile.tex &

D) # tron -p rs / -p x /usr/ucb/finger /usr/etc/fingerd -p w /usr/adm/daemon.log
 -c fingerd &

E) # tron -p rx /usr/ucb/finger /usr/etc/fingerd -p rw /usr/adm/daemon.log -c
 fingerd &

F) % ps -ax | egrep fingerd | egrep -v egrep
20960 ? I 0:04 fingerd
% tron_loan 20960 -p r ~/.{plan,project}

could be facilitated by tron_loan. As shown in Example E, the super-user initializes the finger daemon without the pervasive read rights given in the previous example.

Upon logging in, the user can temporarily grant rights to read their .plan and .project to the finger daemon using tron_loan as depicted in Example F. In this example, we have created a normal shell that, while it exists, grants the rights to read the .plan and .project files to the finger daemon. (Obviously, these steps could be automated in the user's .login using, for example, awk.) Another user, attempting to finger a user who is not logged in is denied permission to read their .plan and .project files.

6. Performance Impact

Two benchmarks were run to determine the impact of the TRON domains to existing UNIX performance. The first benchmark determines the time to fork a null thread, the second to open a file, write 1000 bytes, and close it. The results are summarized in Table 1. The table shows average times, in microseconds, for the *forktest* and *filetest* benchmarks, tested without TRON (i.e., standard ULTRIX distribution), with TRON using the full TRON domain, using the stdtron TRON domain with 15 capabilities, using a TRON domain with 50 capabilities, and, finally, using a TRON domain with 80 capabilities. The timing values are averaged over 10,000 trials. In the restricted domain tests, the required capability is found at the end of a linear search.

The results are fairly telling. The cost to verify a capability, even when using an inefficient linear search mechanism, has negligible impact upon system performance in comparison to the larger costs of the system call and I/O.

7. Related Work

Capabilities and access control lists are wellknown topics in operating systems design[6][17]. There have been several papers on the subject of strengthening UNIX security by adding capabilities or access control lists. The most commonly suggested method is to grant users access control over files[8][16][19]. Access control lists of this type have several drawbacks. Since users are persistent, the lists must be persistent and stored on disk in protected files. If one user runs a program owned by another user, functionality and security requirements force both users to have appropriate access control list entries in all relevant files, and possibly delete entries after each use. Users are thus required to perform a lot of maintenance on their access control lists. Furthermore, these systems offer no protection for the user against programs owned by the user, which may contain errors, Trojan Horses, or viruses.

Lampson [14] points out that users must be protected against activities by their own programs as much as by other users. Wichers, et.al. [20], propose an access control system, PACLs, in which rights are granted to programs rather than users. This method protects somewhat against viruses and Trojan Horses, since they must have the correct program name to work their effects. However, their system requires extensive effort by the user. Among other things, special files must be maintained, and the user must enter a password to perform actions such as removing a file or editing the control lists. To save maintenance time, they allow users to create files that are not constrained by PACLs, or to temporarily disable PACLs. The danger is that coupling extensive maintenance requirements with the ability to completely opt out of the security mechanisms can lead to reduced use of these mechanisms.

test	without TRON	full rights	stdtron (15 capabilities)	50 capabilities	80 capabilities
forktest	3388	3438	3625	3699	3719
filetest	33726	33648	33730	33786	33717

Table 1: Performance Benchmarks

Lai & Gray [13] propose a system that is similar to TRON. In their system, processes are granted rights to files. Processes reside in Untrusted Process Families (UPFs), which contain lists of capabilities. Child processes inherit their parents' UPF, or can be placed in a new UPF that contains a subset of the capabilities of the parent's UPF. This is the paradigm used in TRON. Unlike files and users, processes are temporary. Thus, process capabilities are temporary, reducing maintenance on the part of the users. Furthermore, a process is a more fine—grain construct than a program or a user. A single user can simultaneously have more non—interfering "sets" of capabilities at the process level than at the program or user level.

However, the Lai & Gray approach suffers from an inflexible and often incorrect capability determination mechanism, which strongly restricts the types of programs that can be protected. The system grants capabilities to processes based on their argument lists and to temporary files they create. The actions of processes on files named as arguments are unrestricted, which may not be the user's intent. Also, the common use of permanent files other than those passed as arguments is disallowed. Furthermore, program arguments are often not filenames, but in this system must be treated as such. Finally, the approach has no means of indicating that an argument specifies a directory tree, making it impossible to protect programs, such as 1s, find and rm, that recursively access directories[7].

Programs that do not cooperate with the capability determination scheme must opt out of running in a protected domain. To compensate, Lai & Gray propose a cumbersome mechanism for specifying that such programs are "trustable," involving intervention by a system administrator to validate the program, then mark a bit in the program's inode structure. Ultimately, this system suffers from the same problem as PACLs in combining a difficult to use and intrusive protection facility with a method of completely opting out of the system.

8. Contributions

Increasing UNIX security while maintaining current standards of flexibility and openness has been a long-standing research and industry goal. We feel

that the integration of TRON moves UNIX a long way towards meeting this goal.

By using process—based capabilities we restrict the activities on files at a finer granularity than user or program—based capabilities. As Lai and Gray point out in [13], the goals of computer users are effected through process invocation, and therefore it is the user's intended process accesses that should be delineated and enforced. Unlike Lai & Gray's system, however, we leave it to the user to directly convey their intended process accesses. Because we create new protection domains as part of the UNIX forking mechanism (tron_fork), rather than as part of the UNIX exec mechanism, we don't run into their necessity to guess the required capabilities at program execution time.

TRON enables process-based discretionary access control with practically no extra maintenance on the part of the user or system administrator. No changes to the file system or special files are required. While the kernel must be recompiled, all existing user-level programs are binary compatible¹. The new TRON user commands and system calls are available to all users, without requiring super-user privileges.

We feel that our system of naming file capabilities is powerful and correct for directory-tree-based file systems such as UNIX. TRON employs directory capabilities that apply to all files within a directory, and an optional *subtree* right allowing the ready extension of capabilities to entire directory subtrees. This scheme is flexible enough to specify large portions of the file system with just a few capabilities.

Perhaps our most important contribution is that we have created a system that is powerful yet easy to operate and understand. Users can take advantage of the TRON service with a minimal amount of effort. And ultimately, the success of any computer security system is dependant on the willing participation of the system's users.

^{1.} System programs that include proc.h, such as ps, must be also be recompiled in our prototype implementation, but this can be avoided as discussed in the footnote of section 4.3.

9. Conclusion

We have argued that a process–specific, expressive, and easy–to–use service is an effective means of providing protection from Trojan Horses, computer viruses, etc. We have presented the design of TRON as an extension to the UNIX operating system that provides such a service in a safe, transparent and flexible manner. We have successfully integrated this service into an existing UNIX implementation and demonstrated its usefulness. Finally, we have shown how our service relates to previous efforts to strengthen UNIX security and provides additional benefits beyond those efforts.

10. Acknowledgments

We'd like to thank Hank Levy, Jeff Chase, Alex Klaiber, Dylan McNamee and Alec Wolman for their guidance and advice in the development of TRON. Thanks also to Debbie Berman for editing advice. And additional thanks to Paul Barton–Davis for introducing us to the lore of malicious programs.

11. References

- [1] Bershad, B. N. and Pinkerton, C. B., "Watchdogs—Extending the Unix File System," *Computing Systems*, vol. 1, no. 2, pp. 169-88, Spring 1988.
- [2] Candia, T., "Unix Security Myths and Truths," *EDPACS*, vol. 20, no. 6, pp. 8–13, December 1992.
- [3] Carson, M. E. and Wen, Der Jiang, "New Ideas in Discretionary Access Control," *Proceedings: UNIX Security Workshop*, Portland, OR, USA, pp. 35–7, USENIX, August 29–30 1988.
- [4] Davida, G. I. and Matt, B. J., *Unix Guardians: Delegating Security to the User*, USENIX, Berkeley, CA, USA, 1988.
- [5] Denning, Dorothy E., *Cryptography and Data Security*, Addison–Wesley Publishing Company, Reading, MA, 1982.
- [6] Dennis, Jack B. and Van Horn, E. C., "Programming Semantics for Multiprogrammed Computations," *Comm. ACM*, vol. 9, no. 3, pp. 143–155, March, 1966.

- [7] Duff, Tom, "Viral Attacks On UNIX System Security," Proceedings, 1989 Winter USENIX Technical Conference, San Diego, CA, pp. 165–171, USENIX, January 30–February 3, 1989.
- [8] Fernandez, G. and Allen, L., "Extending the UNIX Protection Model with Access Control Lists," *Proceedings* of *USENIX*, San Francisco, CA, USA, USENIX Assoc., Berkeley, CA, USA, Summer 1988, pp 119–132.
- [9] Hardy, N., "The Confused Deputy (or Why Capabilities Might Have Been Invented)," *Operating Systems Review*, vol. 22, no. 4, pp. 36–8, October 1988.
- [10] Heydon, A., and Tygar, J. D., "Specifying and Checking UNIX Security Constraints," *Computing Systems*, vol. 7, no. 1, pp. 91–112, Winter 1994.
- [11] Howard, John H., Kazar, Michael L., Menees, Sherri G., Nichols, David A., Satyanarayanan, M., Sidebotham, Robert N., West, Michael J., "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 51-88, February 1988.
- [12] Kaplan, R., "SUID and SGID Based Attacks on UNIX: a Look at One Form of the Use and Abuse of Privileges," *Computer Security Journal*, vol. 9, no. 1, pp. 73–7, Spring 1993.
- [13] Lai, Nick, and Gray, Terence E., "Strengthening Discretionary Access Controls to Inhibit Trojan Horses and Computer Viruses," *Proceedings* of *USENIX*, San Francisco, CA, USA, USENIX Assoc., Berkeley, CA, USA, Summer 1988, pp. 275–86.
- [14] Lampson, Butler W., "Protection," *Proceedings* of the Fifth Princeton Symposium on Information Sciences and Systems, Princeton University, March 1971, pp. 437–443, reprinted in *Operating Systems Review*, vol. 8, no. 1, January 1974, pp. 18–24
- [15] Levy, Hank, Capability Based Computer Systems, Digital Press, 1984.
- [16] Low, Marie Rose, and Christianson, Bruce, "Fine Grained Object Protection in Unix," *Operating Systems Review*, vol. 27, no. 1, pp. 33–50, January 1993.
- [17] Organick, E.I., "The Multics System: An Examination of Its Structure. MIT Press, Cambridge MA, USA 1972

[18] Seeley, Don, "A Tour Of the Internet Worm," Conference Proceedings 1989 USENIX Technical Conference, USENIX Assoc., Berkeley, CA, USA, pp. 287–304.

[19] Strack, H., "Extended Access Control in UNIX System V—ACLs and Context," *USENIX Workshop Proceedings on UNIX Security II*, Portland, OR, USA, pp. 87–101, USENIX Assoc., Berkeley, CA, USA, August 27–28 1990.

[20] Wichers, D. R., Cook, D. M., Olsson, R. A., Crossley, J., Kerchen P., Levitt, K. N., and Lo, R., PACL's: An Access Control List Approach to Anti-Viral Security, USENIX Assoc., Berkeley, CA, USA.
[21] Wobber, E., Abadi, M., Burrows, M., and Lampson, B., "Authentication in the TAOS Operating System," Proceedings of the 14th ACM Symposium on Operating System Principles, December 5-8, 1993.

Andrew Berman (aberman@cs.washington.edu) is a Ph.D. candidate in the Department of Computer Science and Engineering at the University of Washington, Seattle, Washington. His research interests include algorithm design, data structures, and computer security. He received his A.B. in computer science from Princeton University in 1988.

Virgil E. Bourassa (virgil@cs.washington.edu) is a Ph.D. candidate in the Department of Computer Science and Engineering at the University of Washington, Seattle, Washington. His research interests include computer operating systems and architectures. He joined Boeing in 1988 and now works as a scientist in the Computer Science organization of the Computer Services Division in Bellevue, Washington. He received his BS in electrical engineering from Arizona State University, Tempe, Arizona, in 1987 and his MS in electrical engineering from the University of Washington, Seattle, Washington in 1990.

Erik W. Selberg (selberg@cs.washington.edu, http://www.cs.washington.edu/homes/speed) is pursuing his Ph.D. in computer science from the University of Washington, Seattle, Washington. His research interests include systems issues concerning security, integrity, and authentication, as well as artificial intelligence issues involving multi-agent coordination and planning. Currently he is investigating methods for conducting commerce on the Internet using software agents. He graduated from Carnegie Mellon University in 1993 with a double major in computer science and logic, and received the first Allen Newell Award for Excellence in Undergraduate Research.

NOTES

THEY COME FROM PALO ALTO

Session Chair: Phil Winterbottom, AT&T Bell Laboratories

NOTES

SIFT – A Tool for Wide-Area Information Dissemination *

Tak W. Yan Hector Garcia-Molina

Department of Computer Science

Stanford University

Stanford, CA 94305

{tyan, hector}@cs.stanford.edu

Abstract

The dissemination model is becoming increasingly important in wide-area information system. In this model, the user subscribes to an information dissemination service by submitting profiles that describe his interests. He then passively receives new, filtered information. The Stanford Information Filtering Tool (SIFT) is a tool to help provide such service. It supports full-text filtering using well-known information retrieval models. The SIFT filtering engine implements novel indexing techniques, capable of processing large volumes of information against a large number of profiles. It runs on several major Unix platforms and is freely available to the public. In this paper we present SIFT's approach to user interest modeling and user-server communication. We demonstrate the processing capability of SIFT by describing a running server that disseminates USENET News. We present an empirical study of SIFT's performance, examining its main memory requirement and ability to scale with information volume and user population.

1 Introduction

Technological advances have made wide-area information sharing commonplace. A suite of tools have

emerged for network information finding and discovery; e.g., Wide-Area Information Servers (WAIS) [KM91], archie [ED92], World-Wide Web (WWW) [BLCGP92], and gopher [McC92]. However, these new tools have one important missing element. They provide a means to search for existing information, but lack a mechanism for continuously informing the user of new information. The exploding volume of digital information makes it difficult for the user, equipped with only search capability, to keep up with the fast pace of information generation. Instead of making the user go after the information, it is desirable to have information selectively flow to the user. In an information dissemination (aka. alert, information filtering, selective dissemination of information) service, the user expresses his interests in a number of long-term, continuously evaluated queries, called profiles. He will then passively receive documents filtered according to the profiles. Such a service will become increasingly important and form an indispensable tool for the dynamic environment of wide-area information systems.

A very simple kind of information dissemination service is already available on the Internet: mailing lists (see e.g., [Kro92]). Hundreds of mailing lists exist, covering a wide variety of topics. The user subscribes to lists of interest to him and receives messages on the topic via email. He may also send messages to the lists to reach other subscribers. LIST-SERV is a software system for maintaining mailing lists. A problem with the mailing list mechanism as a tool for information dissemination is that it provides a crude granularity of interest matching. A user whose information need does not exactly match certain lists will either receive too many irrelevant or too few relevant messages. The USENET News (or Net-

^{*}This research was sponsored by the Advanced Research Projects Agency (ARPA) of the Department of Defense under Grant No.MDA972-92-J-1029 with the Corporation for National Research Initiatives (CNRI). The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of ARPA, the U.S. Government, or CNRI.

news) system [Kro92], an electronic bulletin board system on the Internet, is similar in nature to mailing lists. While Netnews is extremely successful with millions of users and megabytes of daily traffic, at the same time it often creates an information overload. Like mailing lists, the coarse classification of topics into newsgroups means that a user subscribing to certain newsgroups may not find all articles interesting, and also he will miss relevant articles posted in newsgroups that he does not subscribe to.

Recent research efforts on information filtering focus on the filtering effectiveness, attempting to provide more fine-grained filtering using relational, rule-based, information retrieval (IR), and artificial intelligence approaches. With few exceptions, they are often small scale (i.e., involving a small number of users or profiles) and thus the need to provide efficient filtering is not apparent. However, in a large-scale wide-area system where the number of information providers and seekers are large, efficiency in the dissemination process is an important issue and must be addressed.

The Stanford Information Filtering Tool (SIFT) is a tool for information providers to perform large-scale information dissemination. It can be used to set up a clearinghouse service that gathers large amount of information and selectively disseminates the information to a large population of users. It supports full-text filtering, using well-known and well-studied IR models. The SIFT filtering engine implements novel indexing techniques, capable of scaling to large number of documents and profiles. It runs on several major Unix platforms and is freely available to the public by anonymous ftp at URL:

ftp://db.stanford.edu/pub/sift/sift-1.0.tar.Z

In this paper we describe SIFT. We present its approach to user interest modeling and user-server communication. We demonstrate the processing capability of SIFT by describing a running server that disseminates tens of thousands of Netnews articles daily to some 13,000 subscriptions. We describe the implementation of SIFT, focusing on the filtering engine. Finally we present an empirical study of SIFT's performance, examining its main memory requirement and ability to scale with information volume and user population.

2 Other Previous Work

Boston Community Information System [GBBL85] is an experimental information dissemination system. Like SIFT, it allows a finer granularity of interest matching than mailing lists or Netnews. Users can express their interests with IR-style, keyword-based profiles. The system broadcasts new information via radio channel to all users, who then apply their own filters locally. While the radio communication channel makes broadcast inexpensive, local processing of mostly irrelevant information is very expensive. (For every user, a personal computer is dedicated for this purpose – this is cited as a major source of complaints from the users.)

Information Lens [MGT+87] provides categorization and filtering of semi-structured messages such as email. The user defines rules for filtering, and the processing is done at the user site. It provides effective filtering, but local processing is expensive for large-scale information dissemination. Similarly, the "kill file" mechanism in certain news reader programs allows the user to locally screen out irrelevant articles. A kill file only removes specified articles from newsgroups that a user subscribes to, but it does not discover relevant articles in other newsgroups. To provide the same kind of filtering power as SIFT would require much local processing. It is more cost-effective to pool profiles together to share the processing overhead.

The Tapestry system [GNOT92] is a research prototype that uses the relational model for matching user interests and documents; filtering computation is done not on the properties of individual documents, but rather on the entire append-only document database. Efficient query processing techniques are proposed for handling this kind of queries. Tapestry is built on top of a commercial relational database system. The Pasadena system [WF91] investigates the effectiveness of different IR techniques in filtering. It collects new documents from several Internet information sources, and periodically run profiles against them. Similar to SIFT, it uses a clearinghouse approach, but efficient filtering is not addressed.

In [YGM94b, YGM94c] we proposed a variety of indexing techniques for speeding up information filtering under IR models. There we evaluated these techniques using analysis and simulation. The SIFT filtering engine is a real implementation of one class of index structures that we found to be efficient.

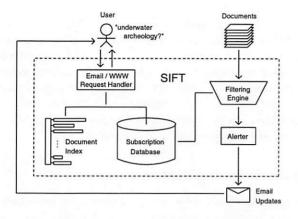


Figure 1: An overview of SIFT

3 SIFT

We begin our presentation of SIFT with an example. Suppose a user is interested in underwater archeology (Figure 1). He sends an email subscribe request to a SIFT server specifying the profile "underwater archeology," and optionally parameters that control, for example, how often he wants to be updated or how long his subscription is for. He may alternatively access the SIFT server via WWW, using a graphical WWW client interface to fill out a form with the subscription information. (Before he subscribes, he may test run his profile against an index of a sample document collection.) His subscription is stored in the subscription database. As the SIFT server receives new documents, the filtering engine will process them against the stored subscriptions, and notifications will be sent out based on the user-specified parameters.

In the following we detail the SIFT system from the perspectives of user interest modeling and communication protocol. We then illustrate SIFT using the Netnews SIFT server.

3.1 User Interest Modeling

A user subscribes to a SIFT server with one or more subscriptions, one for each topic of interest. A subscription includes an IR-style profile, and as mentioned additional parameters to control the frequency of updates, the amount of information to receive, and the length of the subscription. A subscription is identified by the email address of the user and a subscription identifier.

3.1.1 Filtering Model

The interest profile can be expressed in one of two IR models: *boolean* and *vector space* [Sal89]. We first focus on the vector space model.

In vector space model, queries and documents are identified by terms, usually words. If there are m terms for content identification, then a document D is conceptually represented as an m-dimensional vector $D = \langle w_1, w_2, \ldots, w_m \rangle$, where weight w_i for term t_i signifies its statistical importance, such as its frequency in the document. We may also write D as $\langle (t_{i_1}, w_{i_1}), \ldots, (t_{i_k}, w_{i_k}) \rangle$ where $w_{i_j} \neq 0$; e.g., \langle (underwater, 60), (archeology, 60) \rangle . A query is similarly represented. For a document-query pair, a similarity measure (such as the dot product) can be computed to determine how "similar" the two are. In an IR environment, the top ranked documents are retrieved for a query.

In an information filtering setting, a key question is how many documents to return to the user. We could have allowed the user to receive a fixed number of top ranked documents per update period, e.g., as done in [FD92]. However, this is not very desirable: in a period when there are many relevant documents, he may miss some (low recall¹); in a period when there are few interesting documents, he may receive irrelevant ones (low precision¹). We instead allow the user to specify a relevance threshold, which is the minimum similarity score that a document must have against the profile for it to be delivered. A default value is supplied to the user for convenience.

Instead of using the vector space model, the user may use boolean profiles to specify words that he wants in documents received, and words to be excluded. For example, the boolean profile "fly fishing not underwater" is for documents that contain both words "fly" and "fishing" but not the word "underwater." The reader may note that the SIFT boolean model only allows conjunction and negation of words. However, the user may approximate disjunction semantics by submitting multiple subscriptions (though a document may match more than one subscriptions).

3.1.2 Profile Construction and Modification

To assist the user with the construction of a profile, a SIFT server provides a test run facility. The user may

¹Recall is the proportion of relevant documents returned, and precision is the proportion of documents returned that are relevant.

run his initial profile against an existing, representative collection of documents to test the effectiveness of his profile. He may interactively change the profile and (for weighted profiles) adjust the threshold to the desired level of precision and recall. When he is satisfied with the performance of the filtering, he may then subscribe with the selected settings.

After the user receives some periodic updates, he may decide to modify his profile or change the threshold for vector space profiles. He may do this by accessing the SIFT server. Furthermore, for vector space profiles, relevance feedback [Sal89], a well-known technique in IR to improve retrieval effectiveness, can be used. The user simply gives SIFT the documents that he finds interesting; after examining them, the server adjusts the weights of the words in the user's profile accordingly.

3.2 Communication Protocol

There are two modes of communication between the user and a SIFT server. In the interactive mode, the user subscribes, test-runs a profile, views, updates, or cancels his subscriptions. In the passive mode, the user periodically receives information updates. Instead of developing a new communication protocol, we make use of current technologies: email [Cro82] and World-Wide Web HTTP [BLCGP92].

First we discuss the interactive communication mode. Email communication is the lowest common denominator of network connectivity. By having an email interface, a SIFT server is accessible from users with less powerful machines, with limited network capability, or behind Internet-access firewalls. We adopt the LISTSERV mailing list syntax wherever possible, to ensure that minimal learning is required. We also use default settings to reduce the complexity of email requests for novice users.

As the appeal of hypermedia navigation and the development of sophisticated client interfaces are making WWW the preferred tool for wide-area information sharing, we also developed a WWW access interface for SIFT. Using a WWW client program, the user interacts with the SIFT server through a user-friendly graphical interface. We believe the dual email and WWW access covers the tradeoff between wide availability and ease of use.

In the passive mode of user notification, a SIFT server sends out email messages that contain excerpts of new, potentially relevant documents (certain number of lines from the beginning, as specified by the

user). After the user reads the excerpts, he may access the SIFT server to retrieve the entire documents. Currently the excerpts are formatted to be read from regular mail readers. We plan to offer the option of formatting them in HTML, so that the user may view the notifications from sophisticated WWW viewers, and then interactively retrieve interesting documents from SIFT or provide feedback via HTTP.

3.3 SIFTing Netnews

Using SIFT, we have set up a server for selectively disseminating Netnews articles (text articles only; binary ones are first screened out). Like any other SIFT server, the user accesses the Netnews SIFT server via email or WWW. The reader is encouraged to try it out: for email access, please send an electronic message to netnews@db.stanford.edu, with the word "help" in the body; for WWW access, please connect to http://sift.stanford.edu. In February 1994, we publicized the Netnews server in two newsgroups; within ten days of the announcement, we received well over a thousand profiles. The number of profiles keeps increasing and now (November 1994) exceeds 13,000, submitted by users from almost all continents. Table 1 shows some interesting statistics obtained from the server on the day of November 10, 1994. The average number of articles is over the week of November 4 - 10, 1994.

Number of subscriptions	13,381
Number of users	5,146
Daily average number of articles	45,127
Average notification period (in days)	1.4

Table 1: Netnews SIFT server statistics

Apparent from these numbers is that the load on the SIFT server is very high. It is necessary to match an average of over 45,000 articles against some 13,000 profiles and deliver most updates within a day. The efficient implementation of the SIFT filtering engine enables the job to be done on regular hardware, a DECstation 5000/240. Even though we believe SIFT is efficient, there is a limit to the load that it can handle. It is necessary to replicate the server; in fact, we have been in contact with several sites (in the U.S. and Europe) that expressed interests in providing the same service.

The Netnews SIFT server is not meant to be a replacement of the Netnews system, which is an invaluable tool for carrying on distributed discussions. Rather, it provides a complementary filtering service. Only the beginning lines of matched articles are sent out. The user may, after determining that an article is relevant, access his own local news host to access the article. In cases where he does not have access to a news host, he may request the whole article be sent from the SIFT server.

Sifting Netnews is just one application of SIFT. We have set up another SIFT server, disseminating Computer Science Technical Reports (email access: elib@db.stanford.edu, WWW access will soon be available). This server has close to 1,000 subscriptions (November 1994).

4 Implementation

SIFT is implemented in C and has been compiled on several Unix platforms: DEC Ultrix 4.2, HPUX 8.07, and SunOS 4.1. In the following we first give an overview of the components of the system and then focus on the implementation of the filtering engine.

4.1 System Components

The program mailui implements the email request handler shown in Figure 1. It parses email messages, assuming the format in [Cro82]. User requests are processed by accessing the subscription database and the document index (for test runs).

The program wwwi is a so-called "cgi-script" for use with the HTTP daemon released by National Center for Supercomputing Applications. It accepts requests submitted by users using a WWW client (with a form-filling graphical interface), and, similar to mailui, it accesses the subscription database and document index to process the requests.

The program filter implements the filtering engine. It accepts as input a list of documents (in the form of Unix path names) and matches them against the stored profiles. The implementation details are discussed in the next subsection. The output from filter is a file of matchings.

After the filtering is completed, the matchings are sorted by users and subscription identifiers. This is necessary because SIFT sends out updates on a per subscription basis. After sorting, the program alert can be used to send out the message one by one, using Unix sendmail. Included in the messages are excerpts from the matched documents (a few lines from the beginning of the document).

To provide the test run capability, we need to index a collection of documents. This can be done by some publicly available software, and we choose to use the index engine in the WAIS distribution (called waisindex). From our experience it is a very robust implementation. However, since we use a different scoring scheme than WAIS, we made slight modifications to the WAIS code to make it compatible with our filtering engine (i.e., giving the same results). The major changes are to support the use of relevance threshold and the processing of boolean queries. The modified waisindex code is included in our SIFT distribution.

4.2 Filtering Engine Implementation

In IR, to efficient process queries against a collection of documents, an *inverted index* [Sal89] of documents is built, which associates a word with its occurrences in the documents. In the information filtering scenario, we may use the same approach: an index is built of a batch of new documents, and profiles are treated as stored queries that are run periodically. This is the approach used in an earlier SIFT prototype. We refer to this as the document index approach. This approach may not be very appropriate with a high volume of new information. The document index is large and resides on disk. Running a large number of profiles against it requires a lot of disk I/Os.

In the current SIFT implementation, we use a different approach. The idea is to consider information filtering as the dual problem of IR, and treat profiles as documents and documents as queries. Instead of building a document index, we build a profile index. In [YGM94b, YGM94c] we propose and analyze a variety of profile indexing techniques for the vector space and boolean models respectively. In our SIFT implementation, we have chosen good indexing techniques for the two models that are compatible in nature and combined them together in the same index structure. Below we briefly present this combined index structure through an example. For a detailed description, as well as descriptions of other profile indexing techniques and analysis, the reader is referred to [YGM94b, YGM94c].

Referring to Figure 2, suppose P1 is a weighted profile ((underwater, 60), (archeology, 60)) and P2 is a boolean profile "fly fishing not underwater." For each word, we associate it with an *inverted list* of *postings* that reference profiles containing it. For example, in

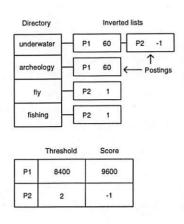


Figure 2: Profile index and auxiliary data structures

Figure 2 the inverted list for word "underwater" is made up of two postings referencing P1 and P2. In the posting, if the profile is weighted, we include the weight of the word in the profile; e.g., 60 for "underwater" in P1. If the profile is a boolean one, we assign a weight of 1 to a non-negated word, and -1 to a negated one.

In addition, we make use of two arrays called Threshold and Score; each profile has an entry in each array. For each profile, its Threshold entry stores the minimum score that a document must have against it to be a match. For a boolean profile, this minimum is equal to the number of non-negated words. For a weighted profile, the minimum is derived using the relevance threshold specified by the user. The Score entry is used to accumulate the score that a document has against the profile.

Now suppose a document D containing the words "underwater archeology" (and none of the other profile words) is being processed. Suppose the vector space representation for D is ((underwater, 80), (archeology, 80), ...); i.e., the words "underwater" and "archeology" are both assigned weights of 80. To process the word "underwater," we look up the directory and access its inverted list. For a weighted profile like P1, the document weight of "underwater" is multiplied with the weight in the posting (60), and the product is accumulated in P1's Score entry (note that we are calculating the dot product of the two vectors). For a boolean profile like P2, we simply add its posting weight (-1 for "underwater") to the Score entry. Similarly the word "archeology" is processed. Finally, profiles whose resulting Score entry is no less than the Threshold entry match the document, such as P1 in Figure 2.

5 Performance Study

A practical challenge for SIFT is to process a new batch of documents within the smallest time unit of notification. For example, in the Netnews SIFT server, it is critical that the filter and notify process is finished within 24 hours. Thus, it is important to understand and characterize the performance of the SIFT system, and measure how it scales with the number of profiles and volume of information.

First we study the performance of the processing time with respect to the number of documents and number of profiles. We then compare the performance of the filtering engine with and the alternative document index approach discussed in Section 4.2. Finally we investigate the main memory requirement of the filtering engine. All experiments are run on a dedicated DECstation 5000/240 running Ultrix 4.2 and all times reported are real (wall clock) time.

The study was performed in early July, 1994. The test data consisted of 38,000 articles received on the day of July 6, 1994 by our department's news host and 7,000 randomly selected subscriptions from the Netnews SIFT server at that time. The average article size was 269 words (a word is defined as an alphanumeric string longer than 2 characters). The subscriptions were stored on a local disk, while the news articles were stored on an NFS-mounted disk (from the news host). We repeated the experiments with test data from two other days and the results were within ±10% of those reported below.

In the result graphs that follow, we divide SIFT processing time into these four steps:

- 1. Build Time The profile index structure is built. Other auxiliary data structures are allocated and initialized.
- 2. Filtering Time Documents are run against the index one by one. Document-profile matchings are written into a file.
- 3. Sorting Time The document-profile matching file is sorted by user email address and subscription identifier, using Unix sort command.
- 4. Notify Time The sorted matching file is read. Excerpts of matchings for each subscription are prepared into an email message. Unix sendmail is invoked to send out each message.

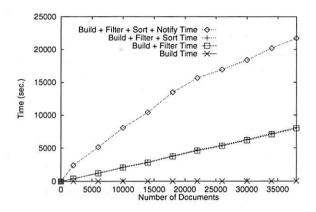


Figure 3: SIFT performance vs. # documents

5.1 Coping with Information Volume

First we investigate the relationship between the processing time and the number of documents. Figure 3 shows the results of processing the 38,000 articles against the 7,000 subscriptions. The time it takes to build the profile index is very small and is as expected independent of the number of documents. The filtering time is linear with respect to the number of documents, with slope 0.21 sec./document (value obtained by curve-fitting using Mathematica). The time it takes to sort the matching file is negligible. The proportionality constant for the total running time is 0.63 sec./document. (Recall that the articles are stored on a remote news host, so some fraction of this time is spent in reading the articles via local network. This is confirmed by looking at the CPU utilization, which is found to be 42.8%.)

To measure the notify time, since it would be impossible to send the same updates repeatedly to our real users, the notify times shown in Figure 3 (except the rightmost data point) are interpolated results. We assume that the time it takes to send out notifications is proportional to the number of matchings in the matching file. We have verified this assumption with a separate experiment and derived the proportionality constant. Then in the experiment for Figure 3, we count the number of matchings in the matching file at the data points shown. We compute the notify time as the product of the number of matchings and the proportionality constant. The results show that a large fraction ($\approx 63\%$ for 38,000 documents) of the total time is spent in sending out subscriptions.

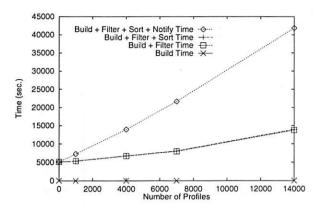


Figure 4: SIFT performance vs. # profiles

5.2 Coping with Subscription Growth

In the second experiment, we look at the relationship between the processing time and the number of profiles. We repeat the process of filtering 38,000 documents against 1, 1,000, 4,000, 7,000, and 14,000 subscriptions. The last run is done simply by duplicating the 7,000 subscriptions in the database. Figure 4 shows the results. Again, the profile index build time is negligible. For the filtering time, we find that even for one profile, it takes 5,105 sec. to filter. This is the time spent reading in the articles via NFS. Beyond the initial start-up cost, filter time is apparently linear to the number of profiles, with slope 0.62 sec./profile. The notify time is similarly obtained as discussed above. We find the total running time to be linear with respect to the number of profiles, and the slope is 2.63 sec./profile.

5.3 Performance Improvement

To quantify the performance improvement obtained by using the filtering engine, we compare it with the earlier SIFT implementation which runs profiles against a document index to perform the filtering. This process consists of these three steps:

- 1. Build Time The document index structure is built.
- Filter Time Profiles are run against the document index one by one. Document-profile matchings are written into a file.
- 3. Notify Time Update messages are sent out.

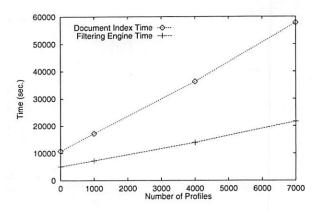


Figure 5: Comparing performances of document index and filtering engine

Comparing this with the filtering engine process, we note that although the first steps have similar names, they represent totally different work. The document index takes much longer to built since there are many more documents than profiles and a document is much larger than a profile. Also the sorting of matchings is not needed in this approach because the filtering is done on a per subscription basis. In fact, a notification may be sent out as soon as a subscription is processed. However, for comparison purposes we opt to separate the work into the shown sequence.

First we compare the total running times of the two approaches. Figure 5 shows the results of processing different number of profiles against 38,000 documents. The slope for the document index total running time is 6.68 sec./profile, compared with 2.63 sec./profile for the filtering engine. If we focus on the 7,000 profile runs, the total running time with the document index is 57,745 sec., compared with 21,652 sec. with the filtering engine. Thus, the SIFT filtering engine is more than twice as fast as the document index approach.

Figure 6 shows the time breakdown for the document index approach. We see that the document index build stage makes up a significant portion (10,803 sec.) of the whole process. The majority of the time is spent in the second stage; for the 7,000 profile run, the filter time is 33,345 sec. or 58% of the whole processing time. On the other hand, the notify time now takes up a smaller fraction of the running time.

It may be argued that the document index build time should also be included in computing the total running time for the filtering engine case, since it

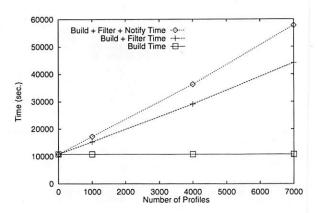


Figure 6: Breakdown of running time for filtering using document index

would be built in that case to provide the test run facility. However, if the document index is used strictly for test run purpose, we might simply build an index with any representative collection and not once for every batch. Even if we do include the document index build time, the total time for 7,000 profiles sums up to 32,455 sec., still just 56% of that for the document index approach.

5.4 Main Memory Requirement

By inspection of the filtering engine code, the formula for main memory requirement (in bytes) for the SIFT filtering engine is:

```
Main memory requirement =
# profiles × 132 +
# distinct profile words × 32 +
# profile word occurrences × 12
```

The first term on the right-hand-side is for the auxiliary data structures (132 bytes per profile; besides keeping the Threshold and Score entries, we also need to keep the email address and other information in main memory to efficiently output the matchings). The second term is for the directory nodes (one per distinct profile word). The last term is for the posting nodes (one per profile word occurrence). Using the subscriptions to the Netnews SIFT server, we count the various numbers on the right-hand-side of the above formula and compute the main memory requirement for various number of subscriptions. The results are shown in Table 2.

# profiles	# distinct words	# total words	Main memory required (bytes)
1,000	1,595	2,334	211,048
2,000	2,806	4,756	410,864
3,000	3,746	6,893	598,588
4,000	4,657	9,244	787,952
5,000	5,535	11,735	977,940
6,000	6,278	14,102	1,162,120
7,000	7,001	16,489	1,345,900

Table 2: Filtering engine main memory requirement

Several interesting statistics can be obtained from these data. The total number of word occurrences is linear with respect to the number of profiles, and the slope is found by curve-fitting to be 2.34 words/profile. On the other hand, the number of distinct words increases more rapidly in the beginning, but at large number of profiles, seems to approach the slope 1 word/profile. Making the rough assumption that the number of distinct words is also linear with the number of profiles, we use curve-fitting to derive the ratio (main memory requirement / number of profiles) and find it to be 195 bytes/profile. This translates to a capacity of 5,128 profiles per megabyte. Thus, with today's large memories, SIFT can process hundreds of thousands of profiles.

6 Conclusion

The user population of our SIFT Netnews server has grown at a rate of approximately 1,400 subscriptions a month; this indicates that there is a recognized need for wide-area information dissemination. Further, since the server was made public, we have received a lot of feedback and comments, which are predominately positive. The server was reported in numerous magazines and newsletters (e.g., WIRED, Internet World, MicroTimes), resulting in the continued growth of its population. A number of sites have requested the SIFT code, which is now publicly available by anonymous ftp. Anticipating further subscription growth for the service, we are in our initial steps to replicate the server at other sites.

One of the main suggestions for improvement is to provide more expressive filtering models to screen out irrelevant information. Netnews is an extremely diverse source of information, with topics that are extremely disparate in nature. Thus the noise level of the filtered results is in some cases high. In the server for disseminating Computer Science Technical Reports, we found the noise level to be much lower. Still we have taken steps to provide more filtering capability, and the boolean model has been added as a result. Extensions to phrases and proximity matching are also planned.

Evident from the SIFT Netnews server, the efficiency of the filtering process is critical in providing such a service. We believe the indexing method implemented in SIFT scales to large number of users and high information generation rate. The time taken to process a batch of documents against a collection of subscriptions is linear with respect to the number of documents and the number of subscriptions. The proportionality constants are roughly half of those based on a document index.

The sending out of email notifications consumes a major portion of the SIFT running time. To improve this, a more sophisticated document delivery scheme is needed. One idea is to group users geographically [YGM94a]. If a document matches one of the users in a group, a single copy of the document is sent to the group. A local distribution site for the group receives the copy and retransmits it to users locally. This is beneficial if users have overlapping interests, and is also useful in reducing wide-area network traffic. However, to install such a scheme, cooperation from the user side is needed.

Acknowledgements

Thanks to Anthony Tomasic for suggesting filtering Netnews as an application of SIFT and for providing comments to improve the readability of this paper.

References

[BLCGP92] T. Berners-Lee, R. Cailliau, J.-F. Groff, and B. Pollermann. World-Wide Web: The information universe. *Electronic Networking: Research, Applications, and Policy*, 1(2):52-8, 1992.

[Cro82] D. Crocker. Standard for the Format of ARPA Internet Text Messages (RFC 822). Network Information Center, SRI International, Menlo Park, California, 1982.

[ED92] A. Emtage and P. Deutsch. Archie: An electronic directory service for the Inter-

- net. In Proc. Usenix Winter 1992 Technical Conference, pages 93-110, 1992.
- [FD92] P.W. Foltz and S.T. Dumais. Personalized information delivery: an analysis of information filtering methods. Communication of the ACM, 35(12):29-38, 1992.
- [GBBL85] D. Gifford, R. Baldwin, S. Berlin, and J. Lucassen. An architecture for large scale information systems. In Proc. Symposium on Operating System Principles, pages 161-70, 1985.
- [GNOT92] D. Goldberg, D. Nichols, B. Oki, and D. Terry. Using collaborative filtering to weave an information tapestry. Communication of the ACM, 35(12):61-70, 1992.
- [KM91] B. Kahle and A. Medlar. An information system for corporate users: Wide Area Information Servers. Connexions - The Interoperability Report, 5(11):2-9, 1991.
- [Kro92] E. Krol. The Whole Internet User's Guide & Catalog. O'Reilly & Associates, Sebastopol, California, 1992.
- [McC92] M. McCahill. The internet gopher protocol: A distributed server information system. Connexions The Interoperability Report, 6(7):10-14, 1992.
- [MGT+87] T. Malone, K. Grant, F. Turbak, S. Brobst, and M. Cohen. Intelligent information sharing systems. Communications of the ACM, 30(5):390-402, 1987.
- [Sal89] G. Salton. Automatic Text Processing. Addison Wesley, Reading, Massachusetts, 1989.
- [WF91] M.F. Wyle and H.P. Frei. Retrieval algorithm effectiveness in a wide area network information filter. In Proc. ACM SIGIR Conference, pages 114–22, 1991.
- [YGM94a] T.W. Yan and H. Garcia-Molina. Distributed selective dissemination of information. In Proc. Parallel and Distributed Information Systems, pages 89–98, 1994.

- [YGM94b] T.W. Yan and H. Garcia-Molina. Index structures for information filtering under the vector space model. In Proc. International Conference on Data Engineering, pages 337-47, 1994.
- [YGM94c] T.W. Yan and H. Garcia-Molina. Index structures for selective dissemination of information under the boolean model. ACM Transactions on Database Systems, 19(2):332-64, 1994.

Bibliography

Tak W. Yan is a Ph.D. student in the Department of Computer Science at Stanford University, Stanford, California. His research interests are in the areas of database systems, digital libraries, and information retrieval and filtering systems. He received a B.S. in Electrical Engineering and Computer Science from University of California, Berkeley in 1991. In 1993 he received a M.S. in Computer Science from Stanford University.

Hector Garcia-Molina is Professor in the Departments of Computer Science and Electrical Engineering at Stanford University, Stanford, California. His research interests include distributed computing systems, database systems, and digital libraries. He received a B.S. in Electrical Engineering from the Instituto Tecnologico de Monterrey, Mexico, in 1974. From Stanford University he received in 1975 a M.S. in Electrical Engineering and a Ph.D. in Computer Science in 1979.

Performance implications of multiple pointer sizes

Jeffrey C. Mogul, Joel F. Bartlett, Robert N. Mayo, Amitabh Srivastava Digital Equipment Corporation Western Research Laboratory

Abstract

Many users need 64-bit architectures: 32-bit systems cannot support the largest applications, and 64-bit systems perform better for some applications. However, performance on some other applications can suffer from the use of large pointers; large pointers can also constrain feasible problem size. Such applications are best served by a 64-bit machine that supports the use of both 32-bit and 64-bit pointer variables.

This paper analyzes several programs and programming techniques to understand the performance implications of different pointer sizes. Many (but not all) programs show small but definite performance consequences, primarily due to cache and paging effects.

1. Introduction

There is only one mistake that can be made in computer design that is difficult to recover fromnot having enough address bits for memory addressing and memory management [4] (p. 2).

Smaller is faster [11] (p. 18).

Whenever someone has declared that a computer has more than enough addressing bits, time (and not much of it) has proven otherwise. Just when systems with 32-bit addresses have become commonplace, many users have found this inadequate; their problems are too large to conveniently implement using 32-bit pointers. While one can sometimes extend a 32-bit architecture with tricks such as segmentation, ultimately the only efficient and clean solution is to use a large, flat address space. Experience suggests that only power-of-two pointer sizes make sense, and indeed most major vendors are either shipping or planning 64-bit systems.

An increase in address size is a discontinuity: the fraction of "interesting" bits in a pointer shrinks a lot. Programs that were pushing the limits of the old address space consume almost none of the new address space.

While some programs will soon grow to need the new address space, many (if not most) will not. A large fraction of the storage used for pointers often goes to waste. A program that needs 33 bits of address space will not run at all on a 32-bit system, but a program that runs happily on a 32-bit system will "waste" more than half of the bits in every 64-bit pointer.

Does this hurt performance? In this paper, we examine a number of ways in which pointer size affects performance for programs that do not need addresses larger than 32 bits. We will show that the effects depend on many variables, including problem size, intensity of pointer use, memory system design, and luck. In many (perhaps most) cases, performance is independent of pointer size, but sometimes, using too-large pointers can result in extra cache misses, TLB faults, and paging.

64-bit systems not only support larger addresses; they also support larger integer data types (most 32-bit systems already support 64-bit floating-point types). Large integers are clearly useful; for example, a 32-bit unsigned counter that increments every microsecond will overflow in about one hour. We will not, in this paper, examine the performance consequences of large integers, because programmers have understood this issue for many years and because all 32-bit and 64-bit systems allow programmers to control the size of integer variables.

Many programs really do need addresses larger than 32 bits, so 64-bit systems are essential and inevitable. We argue, on the evidence of the experiments reported on this paper, that the best system design gives programmers a choice between 32-bit and 64-bit pointers, on a program-by-program or even variable-by-variable basis.

2. Related work

Microsoft's WindowsTM system supports several different kinds of pointer, because the underlying Intel 80286 memory architecture does not directly support a 32-bit flat address space [15]. (Windows also runs on the Intel 80386, which has a flat 32-bit addressing model, but applications meant to be portable must be compiled to use the more restrictive 80286 model.) 80286 addresses use a 16-bit segment number and a 16-bit offset, and Windows cannot guarantee that data segments are contiguous. Thus, when a program uses a 32-bit pointer, the compiler generates code to extract the segment number and offset, loads the segment register, and then uses the offset to reference the memory object. This makes 32-bit pointers far more expensive than 16-bit pointers, but for reasons unrelated to the issues discussed in this paper. (Programs compiled for the Win32s interface use the 80386 addressing model, and so run more efficiently but cannot be used on 80286 systems.)

Our comparison of applications using 32-bit and 64-bit pointers presupposes that one has the option of using small pointers on a 64-bit system. Some applications do use larger data sets than can be addressed using 32-bit pointers. Moreover, some research projects into operating system design are using 64-bit addresses to accomplish other goals; this means that even the smallest application must use 64-bit addresses. Such systems rely on large addresses; 32-bit pointers simply would not suffice.

Several research groups are examining the use of 64-bit architectures to build single-address-space protected operating systems [6, 7, 27]. In such systems, all protected objects (processes, in particular) share a single address space. The system prevents unauthorized access not by modifying the virtual memory map on each context switch, but by hiding each object at a randomly chosen location in a large, sparsely populated address space. An object's address acts as a capability, since the full address space is too large for a malicious or buggy program to search for an object whose location has not been obtained by proper means.

Carter et al. have proposed building a distributed shared memory (DSM) system using a large address space to give a process direct access to memory objects spread across many nodes [5]. Although most current systems cannot support enough real memory to exhaust a 32-bit address space, the aggregate memory of a large number of such systems could easily exceed the range of a 32-bit address, especially if segmented for convenience in allocation and management.

3. Technical context

In this section, we discuss computer system technology as it relates to pointer sizes. We use Digital's Alpha AXP™ architecture [23, 24], and implementations thereof, as a specific example. In addition to architecture, we look at issues in hardware implementation and program compilation. (Operating system features affect the cost of translation buffer and page faults, but those issues are beyond the scope of this paper.)

3.1. Architecture

The Alpha AXP architecture is a load-store design with flat (unsegmented) 64-bit byte addresses. The architecture does single-instruction loads and stores of properly aligned 64-bit and 32-bit values; other values are accessed with multi-instruction sequences.

All instructions are 32 bits wide; load/store instructions specify memory addresses using a general-purpose (64-bit) register and a 16-bit signed displacement.

The architecture does not specify a single memory-system page size, but rather allows an implementation's basic page size to be 8 KB, 16 KB, 32 KB, or 64 KB. Page table entries may use "granularity hints" to inform the Translation Buffer (TB, sometimes known as a Translation Lookaside Buffer or TLB) that a block of pages can be mapped with a single TB entry; however, in the experiments described in this paper, this feature is not used.

3.2. Hardware implementation

We did the experiments reported on in this paper on several workstations of relatively similar design. Common elements include:

- DECchip 21064-AA CPU [8]
- 8 KB on-chip separate instruction and data caches
 - Physically addressed, direct-mapped, write-through
 - 32-byte blocks
- 4-entry 32-byte/entry on-chip write buffer
- 8 KB page size
- 32-entry on-chip fully-associative data TLB
- 8-entry on-chip fully-associative instruction TLB

- 2 MB board-level cache
 - · Direct-mapped, write-back
 - 32-byte blocks

The specific systems differ in CPU clock rate, main memory size, and disk access time. For the tests reported in this paper, we used three different systems:

- System A, a DECstation[™] 3000/600 (175 MHz clock, SPECint92 = 114.1), with 128 MB of main memory.
- System I, a DECstation 3000/800 (200 MHz clock, SPECint92 = 130.2), with 1024 MB of main memory.
- System N, a DECstation 3000/500 (150 MHz clock, SPECint92 = 84.4), with 64 MB of main memory. (This system has a 512 KB board-level cache.)

3.3. Compilation system

All the programs described in this paper are coded in C, run on the DEC OSF/1® operating system, and were compiled using the standard DEC OSF/1 V3.0 compilers.

The compiler supports several integer data types: "short" (16 bits), "int" (32 bits), and "long" (64 bits). By default, pointers are 64 bits wide. The compiler supports a #pragma statement that allows the programmer to select 32-bit pointers. The programmer can choose a pointer size for an entire module, or for specific declarations within that module. The programmer also specifies at compile time whether the pragma should be interpreted or ignored, so the modified program source can optionally be compiled to use only 64-bit pointers.

For example, this program:

```
char *lp;
#pragma pointer_size (short)
char *sp;
main() {
   printf(
       "sizeof(sp) = %d, sizeof(lp) = %d\n",
       sizeof(sp), sizeof(lp));
}
produces
   sizeof(sp) = 4, sizeof(lp) = 8
```

Normally, the linker arranges programs to start above address 2^{32} ; this means that any attempt to dereference a 32-bit value causes an addressing trap, which aids in the detection of portability problems. A program that uses 32-bit pointers clearly cannot be located above 2^{32} , so at compile time the programmer must tell the linker to use its "truncated address space option," which forces all program addresses to lie below that limit. Of course, this cannot be used with programs requiring more than 2^{32} bytes of virtual address space.

Note that the compiler used in these tests generates essentially the same type and number of instructions for both 32-bit and 64-bit pointers. The *only* difference is that loads and stores of pointer variables read or write 32 bits of memory instead of 64 bits. One could do somewhat better than this for some uses of 32-bit pointers, so our measurements may not reflect the entire potential effect of their use.

Because the programs we tested execute essentially the same instruction stream regardless of pointer size, *all* of the performance effects shown in this paper reflect aspects of the memory system (caches, RAM, TLB, and disk).

4. Results for a contrived program

What aspects of system design interact with pointer size to affect performance? We wrote a simple, contrived program (figure 4-1) to illustrate several of these aspects. We designed the program to emphasize the worst-case effects of using large pointers; it most certainly does not represent real programs, and almost any other program will show smaller effects.

The program simply constructs a circular queue and then follows the pointer chain around the queue. It takes three arguments: N, the number of queue elements; M, the number of iterations (pointer dereferences); and P, the number of references between random "seeks" to a different part of the queue. If P = M, then no random seeks are done (after the first one); if P = 1, then every reference is to a randomly-chosen element.

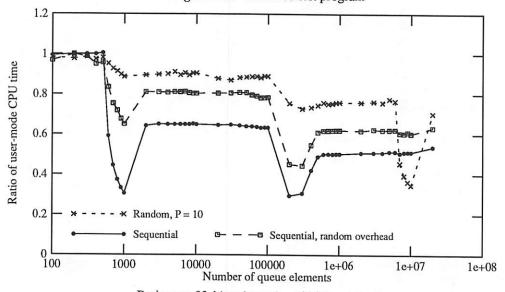
The program can be compiled to use either 32-bit pointers or 64-bit pointers. We ran each version with a pseudo-logarithmic series of values for N, large values of M (at least 10,000,000), and two values of P (P = M, effectively sequential access, and P = 10, almost random access). We measured the user-mode and kernel-mode CPU time, and the elapsed time, for each run.

We plotted the ratios of the times taken by the 32-bit version to the times taken by the 64-bit version; this is more useful than a plot of the actual times, which vary widely. Figure 4-2 shows the ratios of user-mode CPU times; figure 4-3 shows the ratios of kernel CPU times; figure 4-4 shows the elapsed-time ratios.

From figure 4-2 one can see that for small working sets, the pointer size has little or no effect on performance. Once the entire data set no longer fits in the 8 KB on-chip cache, however, the cost of a memory reference increases significantly. Because

```
#include <stdio.h>
                                                          31
                                                          32
                                                                 qp = qpstore;
 3
    #pragma pointer size (short)
                                                          33
 4
                                                          34
                                                                qp->forward = qp;
 5
    struct QueueElement {
                                                          35
                                                                 qp->backward = qp;
 6
      struct QueueElement *forward;
                                                          36
 7
      struct QueueElement *backward;
                                                          37
                                                                 /* make queue elements */
 8
    }:
                                                          38
                                                                 for (i = 0; i < N; i++)
 9
                                                          39
                                                                   /* Insert next element
10
    main(argc, argv)
                                                          40
                                                                      after head of queue */
11
    int argc;
                                                          41
                                                                   qp[i].forward = qp[0].forward;
    char **argv;
12
                                                                   qp[i].backward = &qp[0];
                                                          42
13
                                                          43
                                                                   qp[i].backward->forward = &qp[i];
14
      long N = atoi(argv[1]);
                                                          44
                                                                   qp[i].forward->backward = &qp[i];
15
      long M = atoi(argv[2]);
                                                          45
16
      long P = 1000;
                                                          46
17
      int rval;
                                                          47
                                                                 for (i = 0; i < M/P; i++) {
18
      int i, j;
                                                          48
                                                                   rval = random() % N;
19
      struct QueueElement *qp;
                                                          49
                                                                   qp = &(qpstore[rval]);
20
      struct QueueElement *qpstore;
                                                          50
                                                                   for (j = 0; j < P; j++) {
21
                                                          51
                                                                     /* follow pointer */
22
      if (argc > 3)
                                                          52
                                                                     qp = qp->forward;
23
        P = atoi(argv[3]);
                                                          53
24
                                                          54
25
      qpstore = (struct QueueElement *)
                                                          55
26
                  malloc(sizeof(*qp)*N);
27
      if (qpstore == NULL) {
28
        perror("malloc");
29
        exit(1);
30
```

Figure 4-1: Contrived test program



Ratios are 32-bit pointer time/64-bit pointer time

Figure 4-2: Ratios of user-mode CPU times for contrived program

2000 32-bit pointers (1000 queue elements) fit in 8 KB, but only 1000 64-bit pointers (500 queue elements) fit, the user-time ratio drops precipitously at N = 500. It recovers somewhat at N = 2000 (where even with 32-bit pointers the on-chip cache is too small), but because a cache block holds four queue elements with 32-bit pointers, and only two queue elements with 64-bit pointers, in the sequential case the 64-bit version cache-misses about twice as often.

For the random-access case (P = 10), the cost of frequently generating random numbers partially evens out the ratio of user-mode CPU times. To show this effect, we include the curve in figure 4-2 marked "Sequential, random overhead". This is for a version of the program that calculates a new random value every 10th iteration, as in the P = 10 case, but then ignores this value and runs through the queue sequentially.

The next obvious dip in the user-mode CPU time ratio comes when the working set no longer fits into the 2 MB off-chip cache. This cache holds about 256K queue elements with 32-bit pointers, but only 128K elements with 64-bit pointers.

Figure 4-3 shows the ratios of kernel-mode CPU times, which are rather "noisy" because they include extraneous system activity. These ratios generally track the user-mode ratios until the 64-bit version starts paging, because both programs incur relatively constant kernel-mode overheads of about 1%-2%¹. Once paging begins, the (kernel-mode) CPU cost of handling page faults becomes comparable to the user-mode CPU time. When the problem size is large enough that both programs are paging, the kernel-mode ratio returns to about 50%, since the 32-bit version page-faults half as often.

The elapsed time ratios, plotted in figure 4-4 on a log-log scale, show that cache-miss effects dominate until the problem size exceeds the memory size, at which point paging latency becomes the bottleneck. The ratio drops when the 64-bit version starts to page, then recovers (to approximately 0.55) once the 32-bit version starts paging.

4.1. TLB effects

Each TLB miss causes invocation of a handler that runs with interrupts disabled, and thus does not directly appear in the CPU time statistics sampled by the interrupt clock. We would expect to see TLB miss costs reflected as an increase in the apparent user-mode CPU time. This should appear as the working set exceeds the span of the data TLB (32 entries mapping a total of 256 KB), increasing the TLB fault rate (simulations confirm this). The ratio curves should show a change at 16K queue elements, but there is no such dip in the sequential-case curve in either of figures 4-2 or 4-3. The random-case curve in figure 4-2 does show a small dip. Apparently, the cost of extra TLB misses is insignificant even for this rather stressful program.

4.2. Dependence on data layout

The results we measured for this program depend entirely on how its data structures are laid out. In fact, our first attempt at a test program, which allocated a separate chunk of memory for each queue element (that is, made N calls to malloc() instead of

one call), showed essentially no difference between the 32-bit and 64-bit versions. On this system, malloc(x) always consumes at least 32 bytes of memory for any $x \le 24$, so both the 32-bit and 64-bit versions of the first-attempt program consumed the same amount of space, and both versions put exactly one queue element in each cache block.

4.3. Dependence on code scheduling

The contrived programs described above do nothing but follow pointers, and occasionally generate random numbers. Even the simplest real program usually does some computation on the objects it finds while following pointers. We modified the program in figure 4-1 by adding a float data field to the QueueElement structure; this increased the size of the structure by 50%, for both 32-bit and 64-bit pointers. We modified the inner loop (lines 50-53 in figure 4-1) in two slightly different ways. Version A is:

```
for (j = 0; j < P; j++) {
    qp->data = (i + j + 3.3)/(j + 1.0);
    qp = qp->forward;
}
```

Version B is:

```
for (j = 0; j < P; j++) {
   struct QueueElement *qpn;
   qpn = qp->forward;
   qp->data = (i + j + 3.3)/(j + 1.0);
   qp = qpn;
}
```

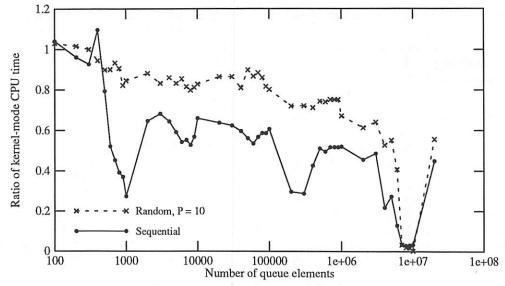
We then ran a set of trials of each program (with P = 10), and plotted the total CPU time (user and kernel combined) in figure 4-5.

The performance of either version of the compute-intensive shows much less dependence on pointer size than does the original, pointer-intensive program (note that the vertical scale in figure 4-5 does not start at zero). This should not be a surprise, since the compute-bound program spends a smaller fraction of its time doing pointer operations.

This only explains part of the difference, however, since version B of the compute-bound program shows a generally closer ratio than does version A (and version B is slightly faster, in absolute terms). Version B manages to bury some of the latency of loading qp->forward, because it can do the arithmetic computation before it needs to use the loaded value. The CPU thus avoids some of the pipeline stall that would otherwise occur. Even for version A the compiler manages to bury much of the load latency; the use of the auxiliary variable qpn simply increases this effect slightly.

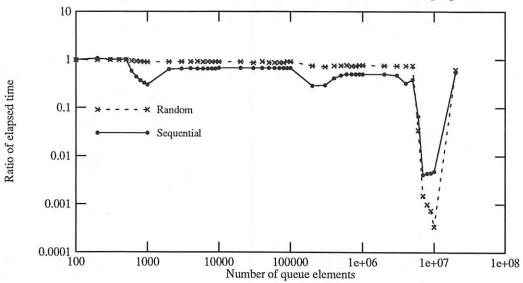
So, a real program may not see some or all of the additional load latency imposed by using larger

¹We believe this overhead is due to device interrupts in progress when the interrupt clock samples the CPU state. This does not necessarily mean that the system spends 1%-2% of its time fielding interrupts [14].



Ratios are 32-bit pointer time/64-bit pointer time

Figure 4-3: Ratios of kernel-mode CPU times for contrived program



Ratios are 32-bit pointer time/64-bit pointer time

Figure 4-4: Ratios of elapsed times for contrived program

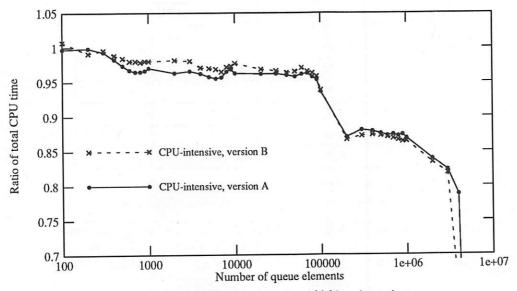
pointers. This depends on the application, the quality of the compiler, and the architecture and implementation of the CPU. The trend in compiler and CPU technology is toward greater tolerance of load latency [9], so one should expect the pointer-size effects to diminish over time. (Store latencies usually do not affect program performance, because modern CPUs and caches can perform stores asynchronously, using mechanisms such as write-buffers and write-back caches.)

4.4. Summary: contrived programs

To summarize what we learned from our contrived programs, with larger pointers:

- Cache misses may increase, as the same number of data items can require more cache lines.
- TLB faults may increase, but probably not enough to worry about.
- Page faults may increase, as the working set increases.

Nevertheless, it is actually rather hard to contrive a program that displays any significant performance dependence on pointer size. Accidents of memory layout, and compiler or CPU techniques to hide load latencies, can reduce or eliminate the cache-related



Ratios are 32-bit pointer time/64-bit pointer time

Figure 4-5: Ratios of total CPU times for compute-intensive program

performance effects. The remaining effect of larger pointers, the earlier onset of paging, afflicts only rather large programs, but may be harder to ameliorate.

5. Results for real programs

Because the results for our contrived program may not represent what happens in real programs, in this section we present measurements for a variety of more-or-less real programs. All tests were run on System A, except where noted.

5.1. Selected SPEC integer benchmarks

The programs in the SPECint92 benchmark suite were intended to be reasonably realistic examples of programs that actual users run. For most of the Clanguage programs in this suite, we compiled and measured versions using both 32-bit pointers and 64-bit pointers. We did not measure gcc, because this program cannot easily be ported to use true 64-bit pointers. Also, we made no attempt to recreate the compiler flags and measurement conditions used for the official SPEC reports; one should not take our measurements as actual SPEC benchmark values.

The SPECint92 C-language programs are [20]:

- compress: A file compression program using Lempel-Ziv coding. The same binary is also used to uncompress files.
- eqntott: Translates a boolean equation into a truth table.
- espresso: Generates and optimizes Programmable Logic Arrays.

- li: Lisp interpreter running a recursive backtracking algorithm to solve the "nine queens" problem.
- sc: A spread sheet program, calculating several different problems.

Table 5-1 reports our results, expressed as the ratio of times for the 32-bit version to times for the 64-bit version. These timings include a little overhead for execution of measurement scripts, so the actual ratios might be slightly larger than the reported ratios (the overhead is less than 1% of the user-mode CPU time, and somewhat larger for kernel-mode CPU time and elapsed time). We report the mean times for at least 10 trials of each benchmark; in some cases, we had to run many more trials, to get run times long enough for accurate measurement.

We include in figure 5-2 a few additional measurements of *compress* and *uncompress* (version 4.0) operating on a larger file than the 1 MB input used in the SPECint92 benchmark. Note that the use of 32-bit pointers slightly hurts performance on *equtott*, *uncompress* and the SPEC-related trials of *compress*, but improves the performance of *compress* applied to a larger file. We ascribe this to cache access patterns, but have not yet confirmed that.

5.2. Late code modification

Compilers typically perform optimizations when compiling individual modules of a large program. One can do certain additional optimizations only on the entire program as a whole. A technique called "late code modification" performs these further optimizations at program link time.

Application	Number of trials	User-mode CPU time ratio	Kernel-mode CPU time ratio		
compress	150	1.008	1.01	1.01	
uncompress	150	1.005	1.00	1.01	
eqntott	10	1.001	0.98	1.00	
espresso	10	0.95	0.98	0.96	
li	10	0.96	0.96	0.96	
sc	10	0.98	0.98	0.98	

Ratios are mean of 32-bit pointer time/64-bit pointer time

Table 5-1: Performance ratios for selected SPECInt92 programs

Application	Input bytes	Output bytes	Number of trials	User-mode CPU time ratio	Elapsed time ratio
compress	3397159	1488027	10	0.993	0.98
uncompress	1488027	3397159	10	1.008	1.05

Ratios are mean of 32-bit pointer time/64-bit pointer time All output directed to /dev/null

Table 5-2: Performance ratios for compression of larger files

OM [25, 26] is an optimizing linker that does late code modification. OM translates the object code of the entire program into symbolic form, recovering the original structure of loops, conditionals, case-statements, and procedures. It then analyzes this symbolic form and transforms it by instrumenting or optimizing it, and generates executable object code from the result. OM makes heavy use of pointers, because its internal representation of a program captures the many relationships between program objects such as procedures, variables, basic blocks, etc.

We ran OM on a number of input programs:

- scixl, a Scheme interpreter with X-library stubs (see section 5.4).
- fea, a finite element analysis tool.
- vcr, a VLSI circuit router.

The *fea* and *vcr* programs are pseudonyms for real programs with substantial market share; for contractual reasons, we cannot give their actual names.

Table 5-3 shows our measurements. We ran trials both on System I, which has a lot of memory, and System A, on which OM's processing of *fea* and *vcr* exceed the available memory. This causes System A to page. (It also increases the elapsed times from minutes to hours, and so we could not run many trials on System A.) The kernel-CPU and elapsed time ratios in this table may be somewhat inaccurate, since we could not easily eliminate other activity during these trials.

Table 5-4 shows how much space OM requires to process each target program. In general, the elapsed time ratios in table 5-3 mirror the size ratios in table 5-4, except in one case. Why, for vcr on System A. do 32-bit pointers outperform 64-bit pointers by so much, in terms of elapsed time? System A has just under 128 MB of real memory; OM's processing of vcr requires just a bit more than that using 32-bit pointers, but quite a lot more using 64-bit pointers. The 32-bit pointer version does far less paging, and so completes much sooner. For both pointer sizes, System A does not page when processing scixl, and pages heavily when processing fea, so for these programs the elapsed-time ratios correspond to the size ratios (although for fea, System A is too slow to be feasible with either pointer size).

These results confirm the lessons of section 4, that smaller pointers usually perform slightly better (and more generally, that performance ratios correspond to size ratios). However, when larger pointers push the working set beyond the size of a cache or real memory, small pointers may show a dramatic advantage.

5.3. Corner-stitching in the Magic CAD system

Many VLSI designers employ the *Magic* CAD system [18] to lay out and process their chips. Most VLSI designs can be expressed as a set of rectangles; Magic represents these rectangles and their positions using an algorithm called "corner-stitching" [17].

Program	Test system	Number of trials	User-mode CPU time ratio	Kernel-mode CPU time ratio	Elapsed time ratio
fea	System I	10	0.94	0.90	0.93
	System A	3	0.92	0.84	0.84
scixl	System I	10	0.95	1.00	0.91
	System A	3	0.96	0.85	0.87
vcr	System I	10	0.93	1.00	0.89
	System A	3	0.91	0.45	0.38

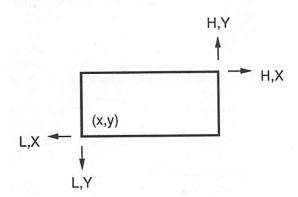
Ratios are mean of 32-bit pointer time/64-bit pointer time

Table 5-3: Performance ratios for OM-optimization of selected programs

Program	Total object file size	Space required using 32b pointers	Space required using 64b pointers		
fea	48080 KB	242360 KB	289264 KB	0.84	0.83
scixl	7792 KB	33448 KB	40488 KB	0.83	None
vcr	27184 KB	128328 KB	155800 KB	0.83	0.34

Table 5-4: OM's space requirements for selected target programs

A Tile, the basic object in the corner-stitching algorithm, has this structure:



The data structure for a Tile stores the (x, y) coordinates of its lower left corner, and four "corner stitches": (L,X), (L,Y), (H,X), and (H,Y). The "stitches" point to neighboring Tiles. For example, the (H,Y) stitch points to the rightmost top neighbor of the Tile. The data structure must also store some information about the nature of the Tile (for example, its layer in the VLSI design).

The minimal representation for a Tile requires four pointers and three integers. Since all coordinates are integral, the integers easily fit into 32 bits. Using 32-bit pointers, a minimal Tile takes 28 bytes; using 64-bit pointers, a minimal Tile takes 44 bytes, although a C compiler would normally pad this to 48 bytes, to naturally align the 64-bit fields within the structure. Thus, the use of 64-bit pointers increases the minimal Tile size by about 70%.

Magic actually adds two pointers, rather than one integer, to the coordinates and stitches, for a total of two integers and six pointers. Therefore, in Magic a Tile requires 32 bytes using 32-bit pointers, and 56 bytes using 64-bit pointers, for a net increase of 75%.

It takes many Tiles to represent a modern VLSI design. For example, BIPS-0, a 32-bit MIPS processor without floating point or virtual memory support [12], required about 3.6 million Tiles. This would need about 112 MB for Tile storage using 32-bit pointers, and 197 MB using 64-bit pointers.

We measured Magic performance on three much smaller designs, a communications interface for a multiprocessor [22], a memory chip [1], and a mesh router chip [10].

We "flattened" the cell hierarchy of the communications interface before running it through Magic; the flattened version requires 208801 Tiles. These Tiles should occupy 6525 KB using 32-bit pointers, and 11419 KB using 64-bit pointers. In fact, to represent and process this design using 32-bit pointers, Magic allocates 9408 KB in addition to its initial memory requirements; using 64-bit pointers, it Thus, the design-specific allocates 13392 KB. memory use increases by only 42%, because Magic allocates quite a bit of non-Tile storage. (Most of the rest of this storage is not actually design-specific, but technology-specific: for example, design rules for the specific CMOS process with which the chip will be fabricated.) The total memory use increases by only 26%, because Magic's other memory use is essentially independent of pointer size.

We used Magic to do a "design-rule check" (DRC) on these three chips. (Design rules specify things such as minimums for rectangle width, spacing, and overlaps.) A DRC spends much of its time following corner-stitch pointers, and doing simple geometric calculations; it should exhibit moderately good locality of reference. In each trial, we had Magic do 50 DRCs of the chip. Table 5-5 shows the mean results over 10 trials, expressed as the ratio of times for the 32-bit version to times for the 64-bit version. The table also shows relative space requirements for all three chips.

For the communications interface chip, the use of 64-bit pointers appears to reduce performance (both user-mode CPU time, and elapsed time) by about 5%. Kernel-mode CPU time shows an even larger change, but has little effect on the elapsed time because it represents only 1.5% of the total CPU time. For the memory chip, pointer size has a smaller effect on user-mode CPU time, probably because far fewer Tiles are used. Kernel-mode CPU time represents a larger fraction (about 5%) of the total CPU time on this problem, so it contributes slightly more to the change in elapsed time.

We believe that the additional kernel-mode time comes mostly from the additional page faults. Almost none of these page faults involve the backing store, because we had plenty of main memory on the test machine. Most are "zero-fill" faults used to add new pages to the address space, and the 64-bit version does about 22% more of these for the communications chip, and about 6% more for the memory chip.

The use of 64-bit pointers appears to impose a relatively small CPU-time cost on Magic. However, for the BIPS-0 design, Magic with 64-bit pointers needs almost 100 MB of additional memory. Put another way, on a workstation with 128 MB of main memory, it would be feasible to design-rule check BIPS-0 only using a 32-bit pointer version of Magic; with 64-bit pointers, the system would page excessively. The most important effect of larger pointers is not the slight increase in CPU time, but the much stricter constraint on feasible problem size.

5.4. Garbage collection in Scheme

Some modern programming languages support "garbage collection" instead of explicit deallocation of dynamic storage. Garbage collection makes programs much simpler to write (one no longer has to worry about forgetting to free a data item, or freeing it too many times, or at the wrong time). It also may improve the performance of code that manipulates

complex structures, since it obviates the need to maintain reference counts or to call explicit deallocation routines. Here we examine the performance of garbage collection in an implementation of the Scheme programming language [2, 19].

Garbage collection complicates the performance picture. (For a full discussion of the cache-related effects of garbage collection, see Reinhold [21].) This implementation uses a "mostly-copying" algorithm [3], which requires that a pool of free space be kept available. The total size of the garbage-collected address space, including live storage and the overhead, is called the "heap." When the heap gets too small, the cost of garbage collection becomes excessive. Even with enough headroom to work in, garbage collection has a run-time cost, which depends somewhat on the size of objects used.

Use of larger pointers can affect the performance of garbage collection:

- 1. It increases the storage-allocation rate (the rate at which address space is consumed). This, in turn, increases the frequency of garbage collection.
- 2. It increases the cost of each garbage collection phase, since this cost is proportional to the amount of live storage.
- It reduces the number of structures that fit into a given amount of real memory, and thus may cause the garbage collector to run out of headroom.

In the worst case, the first and second effects would each increase run-time linearly with the increase in structure size. That is, if a change in pointer size adds 50% to the size of a garbage-collected structure, one would expect the rate of address space consumption to increase by 50%. Assuming that the number of live structures does not change, the amount of data copied during the collection phase would also increase by 50%.

In practice, run-time costs increase less than linearly with pointer size. Programs do things besides consume address space. During garbage collection, the cost of copying an object includes fixed overheads not dependent on object size. Larger objects may show better locality of reference during copying.

The third effect, the loss of headroom available to the garbage collector, cannot be analyzed so easily. For any given application working on a specific problem, some minimum amount of memory is sufficient to support the garbage collector without excessive overhead. When the available memory is less than that amount, garbage-collection costs increase dramatically, and may become effectively infinite.

Design	Total space ratio	Number of tiles	Tile space ratio	User-mode CPU time ratio	Kernel-mode CPU time ratio	Elapsed time ratio	Page-fault ratio
Communications interface	0.79	208801	0.70	0.95	0.93	0.95	0.83
Memory	0.83	55904	0.73	0.96	0.96	0.96	0.87
Mesh router	0.85	25046	0.85	0.96	0.96	0.96	0.90

Ratios are mean of 32-bit pointer value/64-bit pointer value

Table 5-5: Performance ratios for design-rule checking in Magic

(Normally, the garbage collector expands its heap as necessary, but the user can limit the expansion to keep it from overflowing real memory; this would cause the entire program to page excessively.)

We measured the performance of our Scheme system compiled to use either 64-bit pointers or 32-bit pointers for its primary data type (we kept all other pointers at 64 bits). To maintain the required 64-bit alignment for some data objects, the 32-bit version must in some cases pad the pointer fields in its data objects to a 64-bit boundary.

For a sample application, we ran the Scheme->C compiler [2], a Scheme implementation that achieves high portability by using C as its intermediate language, and used it to compile the largest source file in an application called *ezd*. Figure 5-1 shows the ratios of 32-bit times to 64-bit times for elapsed time, garbage collection CPU time, and application (non-GC) CPU time. These tests were run on System N.

One can see from figure 5-1 that the application CPU time depends somewhat on pointer size (use of 64-bit pointers costs about 5%), probably because Scheme programs manipulate pointers heavily. The application CPU time varies little with heap size, because the heap size does not really affect the nature of pointer references made during application execution.

The garbage-collection CPU time depends strongly on heap size, for heap sizes below a threshold: about 7 MB for 32-bit pointers, and about 12 MB for 64-bit pointers. In other words, the use of larger pointers reduces the garbage-collection headroom. Even for heap sizes large enough to reduce the cost of garbage collection below one per cent of the total CPU time, the use of larger pointers seems to add at least 23% to the cost of garbage collection. We believe this results from the increased size of "live" data copied during a garbage collection.

The total elapsed time depends mostly on garbage collection time, for heap sizes below the headroom threshold. For sufficiently large heaps, garbage collection minimally affects elapsed time, and because the application does some I/O, the elapsed time ratio approaches unity.

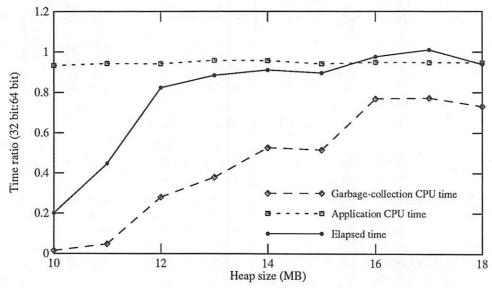
In summary, for this application we found that the use of larger pointers increases the amount of heap (main memory) required for reasonable performance, and slightly increases application CPU time. Large pointers do not significantly affect the running time of the application *if* sufficient RAM is available for the garbage collector; otherwise, they seriously increase elapsed time.

5.5. Sorting

Unnecessarily large pointers could also hurt the performance of pointer-based sorting on large data sets. A sort program spends a large part of its time exchanging the order of records. It can do this by exchanging the records themselves, by exchanging the keys along with pointers to the full records, or by exchanging just the pointers. Pointer sort may be the most efficient technique if the keys are large, especially if the full array of pointers can fit into the CPU board-level data cache. For keys of moderate size, it might be more efficient to use a key sort, which keeps the keys and the pointers together (and so increases locality) [16]. Some phases of a key-based QuickSort can run entirely in a small, on-chip cache [13].

If one is sorting less than a few billion bytes, a sort program has no need for 64-bit pointers. Use of larger pointers than necessary will probably reduce the number of keys (or pointers) that can fit into the CPU's caches, and so reduces the size of the problem that can be sorted without excessive cache-miss overheads.

In an environment that provides only 64-bit pointers, one could implement a pointer-based sort using 32-bit record indices. This requires the execution of several additional instructions each time an index is converted to a pointer, but could be less costly than incurring the extra cache misses imposed by use of 64-bit pointers. It would also require one to maintain separate source-code versions of the sorting program for 32-bit and 64-bit machines.



Ratios are 32-bit pointer time/64-bit pointer time

Figure 5-1: Time ratios for Scheme application

We do not have access to a state-of-the-art sorting program, but we did measure the performance of the UNIX® *sort* command applied to several large files. This sort program manipulates pointers rather than the actual key values. Table 5-6 shows the results; the performance differences of about 5% are consistent with results from other programs.

6. Future work

The results we presented in section 5 suggest a minor but consistent advantage to the use of smaller pointers, in those programs that heavily use pointers but do not need to address a huge data set. However, we would not want to conclude from these results that small pointers are inherently faster. We suggest that future studies should include:

- A wider set of large benchmark applications; the SPEC benchmarks are relatively small, and most of the other applications we measured are not easily suited for use as benchmarks.
- Additional architectures and CPU implementations; we only measured a single CPU implementation of a single architecture, and some of our results may depend on those particulars.
- Multiprocessor and perhaps distributed applications; pointer size could have significantly different implications in such environments.

Programmers have understood for decades that their choice of integer and real variable size can affect performance, and most modern programming languages allow such choices. We do not know of any careful studies quantifying these effects on modern computers, however, but we believe that they could be at least as large, if not larger than, the effects of pointer size.

Given that pointer (and scalar numeric variable) size can effect performance, must the programmer make the choice? Compilers (and optimizers) have freed programmers from many other performance-related decisions. One could imagine a compilation environment (perhaps including link-time optimization) in which the choice of pointer size was deferred until the compiler could determine the necessary range of each pointer variable. For some pointer variables, the compiler might not be able to conservatively infer the necessary size without help from the programmer (in the form of an assertion pragma); in others (for example, a program with constant-sized buffers), the inference might be fairly easy (especially in a type-safe programming language).

7. Summary and conclusions

We have shown that pointer size can affect application performance. The effects depend on pointer-use frequency, address reference patterns, memory system design, memory allocation policy, and other aspects of both program and system, but our results for real programs are consistent with what we learned from a contrived program: larger pointers put greater stress on the memory system, and can greatly affect cache-hit ratios and paging frequency.

Since some applications really do need large pointers, and the performance of other applications

File size	Number of trials	User-mode CPU time ratio	Elapsed time ratio	
9589500 bytes	10	0.975		
95895000 bytes	1	0.948	0.95	

Ratios are mean of 32-bit pointer time/64-bit pointer time All output directed to /dev/null

Table 5-6: Performance ratios for sort program

does not depend on pointer size, we draw the lesson that programmers should use the "right" pointer size for the job. On a 64-bit system, the compiler should give the programmer a choice of pointer size, just as programmers have always had a choice of numeric variable size.

Acknowledgements

We thank Jeremy Dion, Alan Eustace, Jon Hall, Norm Jouppi, and Stefanos Sidiropoulos for their help in preparing this paper. Jim Gray and Chris Nyberg contributed to the section on sorting. Russell Kao, Louis Monier, and David Wall assisted on an earlier draft.

References

- [1] Bharadwaj S. Amrutur and Mark A. Horowitz. Techniques To Reduce Power In Fast Wide Memories. In *Proceedings of the 1994 Symposium On Low Power Electronics*, pages 92-93. San Diego, October, 1994.
- [2] Joel F. Bartlett. SCHEME->C: a Portable Scheme-to-C Compiler. WRL Research Report 89/1, Digital Equipment Corp. Western Research Lab., January, 1989.
- [3] Joel F. Bartlett. *Mostly-Copying Garbage Collection Picks Up Generations and C++*. Technical Note TN-12, Digital Equipment Corp. Western Research Lab., October, 1989.
- [4] C. G. Bell and W. D. Strecker. Computer structures: What have we learned from the PDP-11? In *Proc. Third Annual Symposium on Computer Architecture*, pages 1-14. Pittsburgh, PA, January, 1976.
- [5] John B. Carter, Alan L. Cox, David B. Johnson, and Willy Zwaenepoel. Distributed Operating Systems Based on a Protected Global Virtual Address Space. In *Proceedings of the Third Workshop on Workstation Operating Systems*, pages 75-79. IEEE Computer Society, Key Biscayne, FL, April, 1992.

- [6] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware Support for Fast Capability-based Addressing. In *Proc. 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 319-327. San Jose, CA, October, 1994.
- [7] Jeff Chase, Mike Feeley, and Hank Levy. Some Issues for Single Address Space Systems. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 150-154. IEEE Computer Society, Napa, CA, October, 1993.
- [8] Daniel W. Dobberpuhl, Richard T. Witek, et al. A 200-MHz 64-bit Dual-Issue CMOS Microprocessor. *IEEE Journal of Solid-State Circuits* 27(11):1555-1567, November, 1992.
- [9] Keith I. Farkas and Norman P. Jouppi. Complexity/Performance Tradeoffs with Non-Blocking Loads. In *Proc. 21st International Symposium on Computer Architecture*, pages 211-222. April, 1994.
- [10] C. Flaig. *VLSI Mesh Routing Systems*. TR 35241/87, California Institute of Technology, May, 1987.
- [11] John L. Hennessy and David A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann, San Mateo, CA, 1990.
- [12] Norman P. Jouppi, et. al. A 300Mhz 115W 32b Bipolar ECL Microprocessor. *IEEE Journal of Solid-State Circuits* 28(11):1152-1166, November, 1993.
- [13] Harold Lorin. *Sorting and Sort Systems*. Addison-Wesley, Reading, MA, 1975.
- [14] Steven McCanne and Chris Torek. A Randomized Sampling Clock for CPU Utilization Estimation and Code Profiling. In *Proc. Winter 1993 USENIX Conference*, pages 387-394. San Diego, CA, January, 1993.
- [15] Microsoft Corporation. *Guide to Programming for the Microsoft Windows Operating System* Version 3.1 edition, Redmond, WA, 1992.

- [16] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet. AlphaSort: A RISC Machine Sort. In *Proc. SIGMOD '94*, pages 233-242. Minneapolis, MN, May, 1994.
- [17] John Ousterhout. Corner Stitching: A Data Structuring Technique for VLSI Layout Tools. *IEEE Trans. on Computer-Aided Design* CAD-3(1):87-100, January, 1984.
- [18] John K. Ousterhout, Gordon T. Hamachi, Robert N. Mayo, Walter S. Scott, and George S. Taylor. The Magic VLSI Layout System. *IEEE Design & Test of Computers* 2(1):19-30, February, 1985.
- [19] Jonathan Rees and William Clinger (Editors). Revised³ Report on the Algorithmic Language Scheme. *SIGPLAN Notices* 21(12):37-79, December, 1986.
- [20] Answers to Frequently Asked Questions about SPEC Benchmarks. URL news:2vptki\$kvs@inews.intel.com. June, 1994.
- [21] Mark B. Reinhold. Cache Performance of Garbage-Collected Programs. In *Proc. SIGPLAN '94* Conference on Programming Language Design and Implementation, pages 206-217. Orlando, FL, June, 1994.
- [22] Stefanos Sidiropoulos, Chih-Kong Ken Yang, and Mark Horowitz. A CMOS 500 Mbps/pin synchronous point to point link interface. In 1994 Symposium on VLSI Circuits Digest of Technical Papers, pages 43-44. Honolulu, HA, June, 1994.
- [23] Richard L. Sites (editor). *Alpha Architecture Reference Manual*. Digital Press, Burlington, MA, 1992.
- [24] Richard L. Sites. Alpha AXP Architecture. *Communications of the ACM* 36(2):33-44, February, 1993.
- [25] Amitabh Srivastava and David W. Wall. A Practical System for Intermodule Code Optimization at Link-Time. *Journal of Programming Languages* 1(1):1-18, March, 1993.
- [26] Amitabh Srivastava and David W. Wall. Link-Time Optimization of Address Calculation on a 64bit Architecture. In *Proc. SIGPLAN '94 Conference* on *Programming Language Design and Implementation*, pages 49-60. Orlando, FL, June, 1994.
- [27] Curtis Yarvin, Richard Bukowski, and Thomas Anderson. Anonymous RPC: Low-Latency Protection in a 64-Bit Address Space. In *Proc. Summer 1993 USENIX Conference*, pages 175-186. Cincinnati, OH, June, 1993.

OSF/1 is a registered trademark of the Open Software Foundation, Inc. UNIX is a registered trademark licensed exclusively by X/Open Company Ltd. Alpha, AXP, DECchip, and DECstation are trademarks of Digital Equipment Corporation.

Jeffrey Mogul received an S.B. from the Massachusetts Institute of Technology in 1979, and his M.S. and Ph.D. degrees from Stanford University in 1980 and 1986. Since 1986, he has been a researcher at the Digital Equipment Corporation Western Research Laboratory, working on network and operating systems issues for high-performance computer systems. He is a member of ACM, Sigma Xi, ISOC, and CPSR, the author or co-author of several Internet Standards, an associate editor Internetworking: Research and Experience, and was Program Committee Chair for the Winter 1994 USENIX Technical Conference.

Joel Bartlett received his B.S. and M.S. degrees from Stanford University in 1972. In 1986 he joined Digital Equipment Corporation Western Research Laboratory, working in garbage collection, Scheme, graphics, and PDA's. He is the author of Scheme->C and ezd.

Robert Mayo received his B.S. degree from Washington University in St. Louis, in 1981, and the M.S. and Ph.D. degrees from the University of California at Berkeley in 1983 and 1987, all in computer science. During 1988 he was an Assistant Professor at the University of Wisconsin, but quickly discovered there was no good chinese food in Madison. In 1989 he moved back to the bay area to join Digital Equipment Corporation's Western Research Laboratory. Dr. Mayo's interests include late code modification, computer-aided design tools for VLSI, kung pao chicken, and mongolian beef.

Amitabh Srivastava received a B.Tech. in Electrical Engineering from Indian Institute of Technology, Kanpur, and his M.S. in Computer Science from Pennsylvania State University. Since 1991, he has been a researcher at the Digital Equipment Corporation Western Research Laboratory, working in compilers and link-time code modification. He is the architect of the OM link-time technology which he used to build OM and ATOM systems. Prior to that he was researcher at the Texas Instruments Central Research Labs working on Lisp machines, compilers and object-oriented programming extensions to Scheme. He is the author of the SCOOPS system.

Address for correspondence: Digital Equipment Corporation Western Research Laboratory, 250 University Avenue, Palo Alto, California, 94301 ({mogul,bartlett,mayo,amitabh}@wrl.dec.com)

Idleness is not sloth

Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes Hewlett-Packard Laboratories, Palo Alto, CA

Abstract

Many people have observed that computer systems spend much of their time idle, and various schemes have been proposed to use this idle time productively. The commonest approach is to off-load activity from busy periods to less-busy ones in order to improve system responsiveness. In addition, speculative work can be performed in idle periods in the hopes that it will be needed later at times of higher utilization, or non-renewable resource like battery power can be conserved by disabling unused resources.

We found opportunities to exploit idle time in our work on storage systems, and after a few attempts to tackle specific instances of it in ad hoc ways, began to investigate general mechanisms that could be applied to this problem. Our results include a taxonomy of idle-time detection algorithms, metrics for evaluating them, and an evaluation of a number of idleness predictors that we generated from our taxonomy.

1. Introduction

Resource usage is often bursty: periods of high utilization alternate with periods when there is little external load. If work can be delayed from the busy periods to the less-busy ones, resource contention during the busy periods can be reduced, and perceived system performance can be improved. The low-utilization periods can also be exploited for other purposes—for example, power conservation in a portable system by shutting down parts of it, or eagerly performing work that might be needed in the future.

We call the periods of sufficiently-low system utilization *idle periods*. The definition of "sufficiently low" utilization is application specific; we use the term "idle" even if this utilization is non-zero. During these times the system can execute an *idle task* without affecting time-critical work too much.

The overall structure of the idle-detection framework is shown in Figure 1. External work requests arrive and are executed, requiring resources. Potentially useful idle tasks also consume the same resources. An *idleness detector* monitors the external-work arrivals and the state of the server.

When the detector believes the system will be idle enough for long enough, it starts up the idle task. This executes until it completes, or until the detector signals it to stop—typically when new foreground work arrives. The goal of the detector is to make sufficiently good predictions that the net effect to the system of running the idle task is positive. The best predictions exploit all the idle time, while making no mistaken predictions of idle periods when the system is not, in fact, idle.

There are two basic ways to measure how good an idle-time processing system is. External measures quantify the interference between the idle task and an outside application, and the benefits from running the idle tasks. These measures use units such as additional operation latency or power consumption. Internal measures are based solely on how accurate the detector's predictions are. The external measures are what really matter, but internal measures are useful for guiding the choice of detection mechanism.

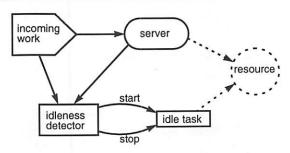


Figure 1: the idle-time processing framework.

The rest of this paper is organized as follows. We begin with a look at several aspects of the problem of making good predictions, including a discussion of external measures and a survey of existing work in this field. We then present an architecture and taxonomy for idleness detectors, and discuss internal measures for evaluating their efficiency. We then use this taxonomy as a tool to generate a suite of detectors, and evaluate their effectiveness under realistic, measured workloads. Some thoughts about opportunities for future work and our conclusions wrap up the paper.

2. Idle-time processing

The purpose of idle-time processing is to improve the system's overall performance during non-idle times. Measuring the improvement requires metrics for the costs and benefits of executing each idle task. These measures vary from one system or application to another. Even within one system, more than one measurement may be appropriate. Often these measures are not directly comparable, or may be subjective in nature. For example:

- The value of powering down a disk drive is the power saved; the cost is that ordinary work may be delayed, extra power consumed during the "recovery" task of spinning up the drive, and the disk lifetime reduced by repeated start-stop cycles.
- The value of disk shuffling (reorganizing data layout on disk) is faster access to frequently-used data. The costs include delaying disk accesses behind a data-move, and buffer-cache pollution.

In addition, the detector itself may intrude on normal system operation: some detectors require significant computation resources to identify idle periods. The work required to determine what to do in the idle time is also a potential resource consumer (for example, disk shuffling is typically based on collected access pattern data; delayed operations are put into a queue that itself consumes resources).

To understand the benefits and costs of idle-time processing, we must first understand how idle tasks can benefit the system, and the ways in which they can create costs: both when executing in an idle period, and if they end up still executing when the system stops being idle. The rest of this section discusses these issues.

2.1. Characterizing idle tasks

Useful idle-time operations fall into a few different categories:

Required work that can be delayed. Examples
include delayed cache writes, migrating objects in
a storage hierarchy, rebuilding a RAID array, or
cleaning a log-structured file system.

- Work that will probably be requested later.
 Examples include disk readahead, eager function evaluation, collapsing chains of forwarding addresses for mobile objects, and eager make.
- Work that is not necessary, but will improve system behavior. Examples include rearranging disk layout, shutting down parts of a system to conserve power, checking system integrity, or compressing unused data.
- Shifting work from a busy to an idle resource.
 Examples include choosing the least-loaded network path, or compressing data when the processor is idle to reduce disk traffic.

Idle tasks can also be characterized by how they react to being stopped and started (we call these *granularity* properties):

- Interruptability: some idle tasks can be interrupted
 at any time, and will stop immediately. Others must
 complete a fixed granule of work before they can
 relinquish the system resources they are
 consuming. For example, a disk write operation
 must run to completion, while a powered-down
 device can be restarted at any time.
- Work loss: when some idle tasks are interrupted, they will lose or must discard some of the work that they have performed. (For example a logstructured file system cleaner may have to abandon work on the current segment.) This cleanup process itself may need resources, and some idle tasks have to be followed by a recovery task to put the system back to a consistent state.
- Resource use: most idle tasks block foreground work from making progress to some degree. In the extreme, they may completely deny foregroundwork access to a resource (e.g., a disk that has been spun down); in other cases, foreground activity simply slows down while the idle task is executing.

Each of these properties affects applications in different ways. For example, high degrees of multiprogramming will probably make a workload more resilient to an idle task that blocks access to a single resource, since there is probably something else useful that can be done while the idle task has the resource.

2.2. Executing idle tasks

Once it has been determined that an idle task should be executed, a number of operations occur, as illustrated by the time line in Figure 2.

Processing begins when the detector signals the idle task to start executing. (Note that there may be a delay between the end of the last piece of ordinary work and the detector emitting its prediction.) The initial activity of the idle task may be to run an initialization step that prepares the system for the main idle task that follows. (For example, it may determine which log-structured file system segments are to be cleaned.) This is followed by one or more executions of the idle task proper. Each of these might take a different amount of time or require a different resource. (Breaking the idle task up in this fashion reduces its granularity, which reduces the cost of a bad prediction.)

Eventually, regular work will again enter the system. If the detector's prediction was accurate, the idle task will have completed execution. If not, the detector signals the idle task to stop, and the task interrupts its activity if possible. It may be necessary to execute a recovery task to bring the system back to normal operation—for example, a powered-down disk must be spun up, or a partially-completed update may need to be undone.

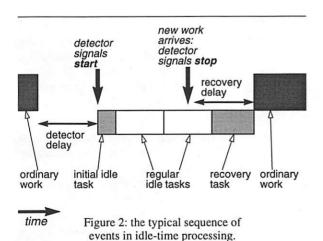
For eager or delayed activity, one must also consider the steps required to delay work, or to detect and use eagerly-performed work later.

2.3. Detecting idle periods

Recall that the *idleness detector* monitors external work requests in order to find idle periods. Figure 3 illustrates the process for one kind of idleness detector.

At the beginning, the system is doing useful work, but then the offered external load decreases below a predetermined threshold. Some time after this happens, the detector makes a prediction about the idle period and signals the idle task to start. Later, as the load increases past the threshold, it signals the task to stop.

More precisely, the detector's problem is to make a series of predictions, each of which identifies the *start time* and *duration* of an idle period. The detector cannot be late with a prediction—otherwise it isn't a prediction. A good prediction will neither start earlier



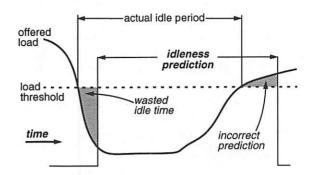


Figure 3: the output from a sample idle detector as the offered load changes.

than the actual start time (so there is no collision between idle processing and ordinary work) nor start much later (which would waste time the idle task could spend doing work); nor will the duration last beyond the end of the actual idle period.

It is usually good to make conservative predictions since the actual cost of incorrectly starting an idle task is usually higher than the opportunity cost of missing a chance to start it successfully.

2.4. Idle task examples

The model of idle-time processing that we have presented so far is rather abstract. To make matters more concrete, we now introduce three examples of idle-time processing in storage systems.

Delayed writeback

The read-write bandwidth of a disk is a scarce resource during a burst of requests. As write buffers increase in size, synchronous read accesses will come to dominate performance in realistic systems because the amount of memory needed to absorb peak write rates is (much) smaller than the quantity needed to cache all reads [Ruemmler93, Bosch94]. The delays seen by reads can be reduced by delaying writes until idle periods, possibly with the help of non-volatile memory [Baker91b, Carson92a].

This is an example of delayed work. When a write operation arrives, it is saved in the cache rather than written to disk. This consumes buffer-cache space, which may reduce the read hit rate in the cache, or require more memory. When the system is idle, these accumulated writes are flushed to disk in groups of N data sectors at a time. (The larger the value of N, the better the requests can be scheduled at the disk [Seltzer90b, Jacobson91].) This flush can potentially delay foreground reads that arrive during the flush. In practice, reads should be given priority over writes [Carson92a]; however, we'll explore the effect of

scheduling the reads and writes with identical priority here.

This idle task requires no special initialization or recovery actions. A good delayed writeback system minimizes read latency and cache utilization. Table 1 summarizes these characteristics.

Initial idle task	(none)
ldle task	flush dirty disk blocks
Recovery idle task	(none)
Granularity	unit: fixed (N sectors) loss: none resources: ties up disk
Measures	max cache space needed change in read latency

Table 1: characteristics of delayed writeback

Eager LFS segment cleaning

In a segmented log-structured file system, blocks are appended to the end of a log as they are written [Rosenblum92, Carson92a, Seltzer93]. The disk is divided up into a number of *segments*, which are linked together to form the log. As blocks are rewritten and their new values appended to the log, earlier copies become garbage that must be reclaimed. A *cleaner* task selects a segment that contains some garbage blocks and copies the remaining valid blocks out of the segment. The segment is then marked as being empty so it can be written over later.

The cleaning operation causes a significant amount of disk traffic, and consumes operating system buffer space [Seltzer93]. The disruptiveness can be minimized if cleaning is performed when there is little ordinary disk traffic. However, segments must be cleaned promptly enough that the system does not run out of clean segments, which would force a segment-clean in the foreground.

The cleaning task is characterized by the delay it imposes on ordinary traffic and how often the system runs out of clean segments. Minimizing the first is best done with an interruptible cleaner that can discard partially-completed operations. Table 2 summarizes these characteristics.

Disk power-down

Several people have investigated powering down disk drives on portable computers to conserve power (e.g., [Cáceres93, Douglis94, Greenawalt94, Marsh93, Wilkes92b]). Using the taxonomy we have developed, the "idle task" is keeping the disk powered off; this is an "optional" task, whose goal is to decrease power consumption; the initial task is to spin the disk down; the recovery task is to spin it back up (Table 3).

Initial idle task	(none)		
ldle task	clean one segment		
Recovery idle task	discard partially-cleaned segment		
Granularity	unit: fixed (1 segment) loss: up to 1 segment, resource: ties up disk		
Measures	foreground cleaning time change in read latency		

Table 2: characteristics of LFS segment cleaning

Initial idle task	spin down disk
Idle task	(none): saves 1.5-1.7W
Recovery idle task	spin up disk: takes 1.5s
Granularity	unit: can be aborted at any time loss: power cost of spinning up disk (3.3J) resource: excludes any other disk accesses
Measures	power saved delay caused to I/O operations

Table 3: characteristics of disk power-down

For example, during normal operation, an HP Kittyhawk disk [HPKittyhawk92] consumes 1.5–1.7W. When it is spun down, it enters a "sleep" mode that consumes very little power. When a disk I/O request arrives, the disk must be powered up, which uses 2.2W for 1.5s (i.e., 3.3J). Power consumption will decrease if the savings from the powered-down mode outweigh the power cost of spinning it up again. For this disk, it can be achieved if the disk is spun down for as little as 2.2s; larger disks take somewhat longer to recoup the spin-up cost. However, spinning the disk down too often will increase the latency of disk requests and increase the chance of disk failure. A good idle-time mechanism will balance these conflicts.

Other examples

There are many possible uses for idle time beyond the three that we have mentioned. Storage, compilation, user interfaces, and distributed systems all exhibit highly variable workloads—a clue that a system could benefit from idle-time processing. Here are a few examples.

Large data structures such as database indices can often be updated quickly at the cost of gradually-worsening access characteristics. For example, nodes can be inserted into or deleted from a binary tree, unbalancing its branches. However, the initial performance can be recovered by a reorganization step (in this case, rebalancing the tree). Such reorganization

tasks are candidates for idle-time processing, thereby moving the cost of rebalancing out of the critical path for updates.

Likewise, some indexing systems in databases segregate the index into two parts: one for "stable" information, for which an efficient hash or tree structure has been built; and one for "unstable" information that has been added or modified recently [Herrin91]. The unstable portion is periodically merged into the stable portion, which involves significant calculation. Doing this in an idle task can make it appear as if it happened "for free".

Compilation is a particularly rich source of idle-time work. One approach that has been explored is to use an eager make facility [Bubenik89], which detects file changes and rebuilds binaries as soon as possible, in the hope that this work will reduce the latency of performing a make when it is asked for later. Costs here include both the processing and disk-resource contention generated by the background compilations, and also the complexity of hiding intermediate results (which may fail) from the user.

Languages such as Cecil, Self, and Smalltalk involve dynamic code generation [Chambers90a]. In many cases this code can be optimized using information generated at run-time, such as the probability of taking a particular branch of a conditional or the inferred type of an object. Modules can be re-optimized when the processor is not busy with normal work during idle time.

There are many instances of distributed system load-balancing schemes that look for "idle workstations" and attempt to use the large number of spare cycles that can be claimed in this way. To keep the workstation owners' happy (and thus the workstations eligible for being used in this fashion), good predictions of when the users are idle are needed [Litzkow88]. Designers of such schemes have used process migration as a mechanism for avoiding loss of work when an idle task is preempted [Litzkow88].

More closely-coupled systems can perform eager data transmission in periods when the network is lightly loaded. For example, release consistency [Carter91] mechanisms can pre-transmit updates to reduce the time to complete a release, which is often on the critical performance path.

The choice of whether to compress data for network transmission can depend on the utilization of both the network and the processor: if the processor is busy, but the network is not, it may make sense to transmit data uncompressed. Likewise, the rate or quality at which multimedia data are played can be varied to take advantage of extra network or processor resource

beyond a guaranteed minimum. These are examples of policy changes as a result of idleness detection.

Forwarding-address chains in a distributed system allow clients to find the current location of a moving object [Shapiro92]. When the object moves, it leaves a pointer to its new location at its old one. Locating the object is done by starting at a place that the object has visited, and following the chain of pointers. Forwarding address chains can be compressed when the system is idle, thereby reducing the time to do future locations.

Disk and file readahead is a common example of eager activity in a storage system [McVoy91, Ruemmler94]. Transparent file compression can improve effective storage capacity, and compressing during idle periods ensures that it does not interfere with ordinary operations. Disk shuffling [McDonald89, Vongsathorn90, Ruemmler91, Akyürek93] involves rearranging the blocks or cylinders on a disk so that frequently-accessed information is located near the middle of the disk with related data close together. Shuffling has non-trivial initial and recovery tasks.

An anonymous reviewer of this paper suggested using idle cycles to perform graphical window-system operations eagerly. For example, hidden or obscured portions of windows could be rendered in idle periods so that exposing them would be quick.

In summary, the existence of burstiness and underutilized resources in a system is an indication that some form of idle-time processing could be exploited. The opportunities are limited mostly by the complexity of deciding what to do when, and by the potential downside of resource contention with foreground activities. Dealing with both of these requires efficient, accurate idleness detectors, whose construction is the subject of the next section.

3. An idle-detector architecture

Having presented an overall framework for idle-time processing, we now turn our attention to how to build idleness detectors. We have found it useful to decompose the problem into a number of separate components, each implementing just one part of the detection algorithm. By combining the parts in different ways we can build detectors on a mix-and-match basis to explore a much wider range of design alternatives than would otherwise be the case.

Figure 4 shows the overall scheme. An idleness detector is composed of a number of predictors and skeptics, along with an actuator:

 A predictor monitors its environment—the arrival process, the server, and possibly other variables such as the time of day—and issues a stream of predictions. Each prediction is a tuple (start time, duration).

- A skeptic [Rodeheffer91] filters and smooths these predictions, possibly combining the results from several predictors.
- The actuator obeys the sequence of predictions it gets as input to start and stop the idle task; its purpose is to isolate the interactions with the idle task from the predictors.

The predictors and skeptics form a directed acyclic graph or DAG with predictors as the leaves and skeptics as the internal nodes. Streams of predictions flow toward the root node, which emits predictions to the actuator.

3.1. Predictors

As with so many problems of this type, optimal idleness detection requires off-line analysis. This is not as useless as it might seem: if usage patterns are stable enough, then a one-time off-line analysis may provide an excellent prediction. For example, "weekends from 1–6 a.m." is a common time to perform system maintenance.

In practice, on-line detectors are of more interest. They can be classified according to the approach used to predict the idle period start time and its duration.

3.1.1. Idle period start time

Simple predictors use little information to make their predictions; more sophisticated ones take advantage of knowledge about the arrival process to make better decisions. We present them here in roughly increasing-complexity order.

 Timer-based: whenever the system runs out of work, the predictor begins a timer. If no work comes in before the timer expires, the predictor declares that an idle period has begun. The timer period can be fixed, variable, or adaptive. A fixed

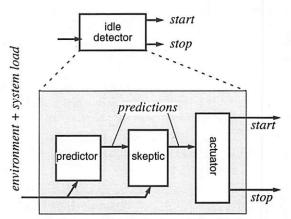


Figure 4: construction of a simple idle detector

period does not change. A variable period is computed as a function of some values in the environment, such as time of day. An adaptive timeout period is increased if predictions are too eager, and decreased if they are too conservative.

- Rate-based: the predictor maintains an estimate of the rate at which work is arriving, and declares an idle period when its rate estimate falls below a threshold. Different threshold rates can be used for "start of idle period" and "end of idle period", thereby providing some hysteresis. Methods for maintaining the estimate include:
 - moving average. The rate is periodically sampled, and the predictor computes a moving average of the samples.
 - event window. The predictor maintains the times of the last n arrivals, and estimates the rate as n divided by the age of the oldest arrival. This is similar to leaky bucket rate-control schemes for high-speed networks [Cruz92].
 - time window. The predictor maintains a list of arrival times more recent than t seconds, and estimates the rate as the length of the list divided by t. This is a variation on the event window method.
- Adaptive rate-based: like rate-based, but with an arrival-rate load threshold that is adapted according to the quality of the predictions.
- Rate-change-based: these predictors maintain an estimate of the first derivative of the arrival rate to predict in advance when the arrival rate will fall below a threshold.
- Periodic: if the workload contains periodic work, a
 digital phase locked loop or DPLL [Massalin89a,
 Lindsey81] can be used to synchronize predictions
 to periodic events in the workload, such as the
 UNIX syncer daemon. By knowing when work will
 arrive, it is also known when the system is idle.

3.1.2. Idle period duration

Duration predictors can use a wide range of techniques to adapt to a changing workload. We list them here according to the amount of information they use about the arrival process.

No duration: no prediction is made (alternately, the
prediction is "forever"). Variants on this include
schemes that try to predict the end of an idle period
as this happens, rather than at its beginning. These
are most useful when the definition of "idle"
allows some residual foreground work. For
example: rate-based; adaptive rate-based; or ratechange-based.

- Fixed duration: a fixed duration is predicted. The simplest form of this is "enough time to run the idle task once."
- Moving average: the predictor keeps a moving (possibly weighted) average of the actual durations. The usual average is the mean, but a geometric average or median can also be used.
- Filtered moving average: as for moving average, but only idle periods greater than some lowerbound are considered during the averaging process.
- Backoff: after each prediction is used, the predictor uses the feedback to determine whether the actual duration was longer or shorter than predicted. If it was longer, the next prediction is increased; if it was shorter, the next prediction is decreased. The increases can be arithmetic, increasing by a constant each time, or geometric, increasing by a factor. The skeptic in Autonet constant [Rodeheffer91] and round-trip timers in TCP [Postel80a, Comer91, Karn91] used geometric increase and arithmetic decrease to maintain a prediction slightly longer than the actual, while an idleness predictor works to keep its predictions slightly shorter.1

The backoff algorithm can be applied either at the end of the prediction period, or at the end of the idle period. The first gives a chance for the algorithm to be much more aggressive in extending its estimates, the latter provides more information, but potentially causes the period to be adjusted much less often.

- Filtered backoff: backoff schemes that only consider actual idle periods longer than a given lower-bound during their backoff calculations.
- Autocorrelation: the autocorrelation on the work arrival process gives the probability of an event arriving or the rate of arrival as a function of time into the future. The predicted duration is the period during which the probability of arrival is below some threshold. The autocorrelation is somewhat expensive to compute, so it might be recomputed periodically rather than continuously. It might also be used to predict the beginning of multiple idle periods.
- Conditional autocorrelation: like a simple autocorrelation, except that multiple autocorrelation functions are computed based on some property of arriving events. For example, the expected future might be different following read requests or write requests.

 Ad-hoc rules: finally, as with predicting the beginning of an idle period, many systems can take advantage of other specific features of the arrival process, such as periodicity.

3.2. Skeptics

A skeptic takes in one or more prediction streams, and emits a new one. They are used to filter out bad predictions and to combine the results from several predictors into a single prediction stream.

Single-stream (filtering) skeptics include:

- low-pass: discards predicted periods that are shorter than some threshold (e.g., the duration of the idle task).
- quality: discards predictions from a predictor that
 is consistently wrong. The skeptic can compute a
 measure of the predictor's accuracy, perhaps using
 a moving average of the measures we propose in
 Section 4, and only pass along predictions when
 the accuracy is above some threshold.
- environmental: discards or modifies predictions according to some external event (e.g., time of day). This can allow idleness predictions to be restricted to times when nobody is around, for example. The time-of-day input can be derived from moving averages of workloads over long periods of time, so this skeptic can be made adaptive.

Perhaps the most important use for skeptics is to combine several prediction streams. For example, a periodic-work detector will not handle non-periodic bursts, while another predictor might. A skeptic could combine the two, only reporting a prediction when both agree.

More generally, a skeptic can combine a number of input streams by weighted voting. Each stream is given a weight, and the skeptic produces a prediction only when the combined weights are greater than some threshold. When the weights are equal and fixed, this becomes simple voting. Alternately, the weights can be varied according to the accuracy of each predictor [CesaBianchi94]. This approach has been shown to yield near-optimal prediction in many cases.

4. Evaluating idleness detectors

A good idleness detector produces a stream of predictions that waste little time at either the beginning or end of an idle period, while rarely making a prediction that starts too early, ends too late, or overlaps much real work. We now describe several measures used to quantify how well a particular detector meets these goals. Remember that their

We describe backoff algorithms with shorthand of the form Arith+/Geom-: this indicates a predictor with arithmetic increases and a geometric decrease policy.

interpretation depends on an application-specified level of acceptable idleness.

We start with several primitive measures:

- The predicted time is the total amount of time a detector declared idle.
- The *actual* time is the total time the best possible off-line detector could produce.
- The overflow time measures the amount of time that the detector declared idle when the system was not idle.
- The *violations* count the number of operations that overlapped the declared idle periods.
- The delay is the sum of the delay imposed on operations, assuming that any operation that starts during a period declared idle must be delayed until the end of the period.
- The intrusiveness of an idleness detector measures how much extra load the detector itself imposes on a non-idle system.

From these basic measures we compute two derived measures:

 The efficiency of a detector: this is a measure of how good the detector is at finding idle periods in the workload. It is defined as

efficiency = predicted / actual

 A detector's incompetence evaluates how much of the predicted idle time is not idle, penalizing overeager detectors. It is defined as

incompetence = overflow / actual

A good idleness detector will have a high efficiency and a low incompetence. Ideally, we'd like to have a single value that give the overall goodness of a detector. However, this is impossible, because the weighting of the value of idle task processing and the costs of incorrect idleness predictions is highly application-specific, and sometimes even subjective at a personal level. Nonetheless, we found it useful to consider three main candidate figures of merit:

1. $merit_1 = efficiency \times (1 - incompetence)$

This merit figure considers only efficiency and incompetence, penalizing efficient detectors that are also incompetent.

2. $merit_2 = efficiency / violations$

This merit figure penalizes a detector heavily for each operation that occurs during a predicted idle period.

3. $merit_3 = efficiency - \alpha \times delay$

This merit figure penalizes a detector for the total time that operations are delayed. The relative importance of delay and efficiency can be scaled by the factor α .

The choice is a function of the application. When spinning down a disk, for example, the power saved is related to detector efficiency, while the cost is the delay to operations. The high subjective impact of the cost suggests a metric that penalizes delays more, such as *merit*₃. Delayed writes, on the other hand, have a less dramatic cost of violations, so *merit*₁ is probably more appropriate.

5. Performance results

To get quantitative measures of the effectiveness of idle-time processing, we used the taxonomy presented in section 3 to design a large number of possible idleness predictor networks. We then implemented the component parts and evaluated how well the networks did in practice. This section reports on what we learned.

5.1. Implementation

We implemented the detectors and idle tasks in the TickerTAIP simulation system [Ruemmler94, Golding94]. In particular, we simulated a host system issuing read and write requests to a set of disks. We used calibrated disk models [Ruemmler94], and exercised our detectors using I/O access traces taken from real systems [Ruemmler93] to avoid making simplifying assumptions about access rates. We used I week subsets of these traces.

We added a few new component classes to the TickerTAIP system. These included an overall framework; multiplexers to distribute events to multiple predictors; the predictors themselves; skeptics; and actuators. Our disk and device driver models were changed to accept detectors. The idleness detectors were connected to the disk devices for taking internal measurements, and to the disk device drivers for external measures.

To perform the evaluations, we introduced the notion of a *busyness detector* to determine when the system should be considered busy (i.e., not idle). This was used to evaluate the idle detectors. Busyness detectors can be application-specific. In practice, the simplest busyness detector was the one used for the results we report here: it declared the system busy if there was at least one request anywhere in it.

We looked at several hundred idle detector networks derived from our taxonomy, evaluating them against our internal measures. We also implemented three skeptics: one that filters out predictions during the busiest six hours of the day (the TODSkeptic); one that filters out predictions from a predictor that has a high rate of violations (MeritSkeptic); and one that

combines predictions from several predictors, outputting only predictions on which a quorum agree (QuorumSkeptic).

5.2. Start-time predictors

To make the presentation of the results less intractable, we picked a fixed duration prediction of 25s and combined this with different start-time prediction algorithms.² The most efficient detectors under this test are:

- Event window, when busy 10% or less over last 5 operations (efficiency = 1.01)
- A 1s fixed timer (1.01)
- Event window with rate <10 IO/s over last 5–25 operations (1.00–0.998)
- Moving average rate of 10 IOs/s or less (0.993)
- Adaptive timer with Geom+/Arith- or vice versa with 100ms increments (0.990)
- Event window, busy 1% or less over last 5 operations (0.985)

The lowest violation rates (violations per second of predicted idle period) come from:

- Event window, rate <0.1 IO/s over 25 operations
- · Same, over last 5 operations
- · Moving average, 0.1 IO/s or less
- · Adaptive timer Arith+/Arith-, 10s increment
- · Adaptive timer Arith+/Geom-, 10s increment
- · 10 second fixed timer

5.3. Duration prediction

The next step is to fix the start-time predictor (we used a moving-average rate-based start-time predictor with a threshold rate of 10 IO/s because that gave pretty high efficiencies). The most efficient predictors under this test are:

- Fixed 25s period (efficiency = 0.993)
- Backoff using Arith+/Arith- in 10ms, 100ms and 1s increments (0.992, 0.990 and 0.988 respectively)
- Backoff using Arith+/Geom- and a 1s increment (0.975)
- A moving average over the most recent durations (0.971)
- Backoff Arith+/Geom- with 100ms increments (0.959), or 10ms increments (0.945)
- Moving average + TOD skeptic (0.759)

From the point of view of violation rates, the following were the best ones:

- Moving average + violation rate skeptic (violation rate = 0.358s)
- Moving average + TOD skeptic (0.722)
- Moving average (0.821)
- Backoff Arith+/Geom- with increments of 10ms (1.02); 100ms (3.61); or 1s (14.6)
- Fixed 25s timer (35.1)
- Backoff Arith+/Arith- with increments of 10ms (36.9), 100ms (44.6), or 1s (48.0).

In some circumstances (such as log-structured file system cleaning), it's useful to have idleness detectors that can reliably predict idle periods of long duration. Under this metric (the mean duration of the idle period predicted), we find the following are best:

- Backoff Arith+/Arith- with increments of 1s (mean duration = 44.8s), 100ms (40.4), or 10ms (34.4)
- Fixed 25s timer
- Backoff Arith+/Geom- with increments of 1s (17.9); 100ms (5.82); or 10ms (2.19)
- Average duration + the merit skeptic (0.982)
- Average duration alone (0.952)
- Average duration + the TOD skeptic (0.944)

5.4. Other combinations

Needless to say, picking the best stand-alone start-time and duration prediction algorithms and combining them doesn't produce the best overall result. Instead, we found (by exhaustive combinatorial analysis!) that the following are among the best pair-wise combinations for efficiency:

- EventWindow, busy 10% or less over 25 events + fixed 25s duration (efficiency = 1.0098)
- The same algorithm over the last 5 events (1.0095)
- Busy 10% or less over 25 events + fixed 25s timer (1.0087)
- Busy 10% or less over 5 events + Backoff Arith+/Arith- with 10ms increment (1.0087)
- Busy 10% or less over 25 events + Backoff Arith+/Arith- with 10ms increment (1.0087)
- Fixed 1s timer, + fixed 25s duration
- Busy 10% or less over 25 events + Backoff Arith+/Arith- with 100 ms increments

Generally, the results were dominated by Event Window predictors with a threshold of busy 10% or less, or 5 IO/s or less, combined with arithmetic backoff. For durations, the best result came from combining these with adaptive timers with 100 ms

² Because 25s is a little shorter than the 30s sync daemon interval on the systems from which the traces were taken.

arithmetic increase and geometric decrease (or vice versa), or by backoff predictors with 10–100ms increment arithmetic increases and geometric decrease.

The best violation rates results came from:

- Adaptive timer Geom+/Geom- + Average duration (violation rate = 0.214/s)
- Moving average 10 IO/s + Average duration + a violation rate skeptic (0.358)
- Event window over 25 events with rate < 0.1 IO/s
 + Average duration (0.362)
- Event window with rate < 10 IO/s and a moving average, fed into a Quorum skeptic (0.373)
- Event window over last 5 events with a rate < 10 IO/s, + Average duration, fed into a violation rate skeptic (0.431)

Generally, rate-based or adaptive Arith+/Arith-timers with 10s increments combined with Average duration predictors or Backoff Arith+/Geom- and 10 ms increments dominate.

Looking at mean duration predictions, we find the longest values come from:

- Adaptive timers, Arith+/Geom- or vice versa, with Backoff Arith+/Arith- (1s increment) duration (mean duration = 340s).
- Adaptive timer Arith+/Arith- and increments of 1s (306s), 10s (288s) or 100ms (256s) + Backoff Arith+/Arith- with 1s increments.

We learned that there can sometimes be unfortunate interactions between adaptive start-time predictors and their duration counterparts. These feedback loops are best minimized by running the start-time prediction algorithm once per actual idle period, and the duration prediction once per predicted idle period, if this is shorter than the actual idle period, but longer than some minimum threshold. (The first prevents interpolicy conflicts; the second allows duration-predictors to respond gracefully to alternating bursts of high activity and long quiet periods.)

5.5. Skeptics

Skeptics proved more important for busy disks, where the workload is more variable than on quiet disks. They are also best at reducing the mean violation rate.

On a busy disk, the TOD skeptic reduces violation rate; in the samples, usually to about half what the same predictor yielded without the skeptic, except when the rate was already low. This generally causes a loss of about 20–30% efficiency, indicating that much of the time that could successfully be declared idle was not in the busy hours of the day anyway.

The Merit skeptic likewise reduces the violation rate. For predictors with a high violation rate, the Merit skeptic reduces violations to less than half of the rate of using the TOD skeptic, but generally at the cost of about half the efficiency. When the violation rate is already low, the Merit skeptic does not give appreciable advantage over the TOD skeptic, and in one case (MovingAverage 10/s, Average duration) increased the violation rate by about 20%

For the Quorum Skeptic, we combined EventWindow and MovingAverage predictors (both at rates < 10/s); the EventWindow and a PLL; and the Moving Average and a PLL. For these combinations, both predictors had to agree on idleness. Using quorums like this leads to short predictions because they all have to agree. We also combined four predictors using a quorum size of two for agreement: two EventWindows with different queue sizes, the EventWindow, and a PLL.

The combination of an event window and a moving average predictor them agree gives a low violation rate (0.23 on a busy disk). The combination of all four gives 99% efficiency and one of the lowest violation rates.

Overall, our top three picks for workloads such as delayed writeback and segment cleaning are:

- EventWindow 10 IO/s + MovingAverage 10 IO/s + Quorum skeptic (both agreeing); gives 0.98 efficiency, 0.23 violations, 0.52s mean duration.
- Four predictors + Quorum skeptic; gives 0.99 efficiency, 0.48 violations, 0.52s mean duration.
- Moving average start-rate 10/s + average duration; gives 0.98 efficiency, 0.49 violations, 1.05s mean duration.

6. Conclusions

The high degree of burstiness observed in many real computer systems gives many opportunities for doing useful work at low apparent cost during idle periods. Many people have observed this, and applied this idea to several specialized domains.

The major contribution of this work is to put the previous approaches into a common framework. In turn, the framework suggested opportunities for combining analysis techniques to improve the quality of idle-time prediction. It also allowed us to propose some figures-of-merit for evaluating idle-time detection algorithms, which we then used to evaluate many different idle-time detectors.

Our framework should be helpful to those looking to exploit low-utilization periods in computer systems, regardless of the precise details of the problem, since the framework itself is independent of the particular domain. Developing the taxonomy was helpful to us in two ways: it improved our understanding of the problem, and it helped us systematize the generation of a large number of potentially interesting detection and prediction algorithms. Without it, we would have had a much harder time exploring the design space.

We were gratified to learn that simple predictors work remarkably well. This is good news: it means that these techniques can be applied successfully in the real world with only moderate effort. Nonetheless, we found that you can't have all three of high efficiency, low violation and long durations, but you can generally get any two.

Acknowledgments

George Neville-Neil suggested predictors based on the first derivative of arrival rate. Fred Douglis's interest in disk power conservation spurred much of our early thinking.

Availability

An extended version of this paper, with more detailed results than we had space for here, is available by anonymous ftp from the directory pub/wilkes on the system ftp.hpl.hp.com.

References

[Akyürek93] Sedat Akyürek and Kenneth Salem. *Adaptive block rearrangement*. Technical report CS-TR-2854.1. University of Maryland, November 1993.

[Baker91b] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-volatile memory for fast, reliable file systems. *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA, 12–15 October 1992). Published as *Computer Architecture News*, **20**(special issue):10–22, October 1992.

[Bosch94] Peter Bosch. A cache odyssey. M.Sc. thesis, published as Technical Report SPA-94-10. Faculty of Computer Science/SPA, Universiteit Twente, Netherlands, 23 June 1994.

[Bubenik89] Rick Bubenik and Willy Zwaenepoel. Performance of optimistic make. Proceedings of 1989 ACM SIGMETRICS and Performance '89 International Conference on Measurement and Modeling of Computer Systems (Berkeley, CA). Published as Performance Evaluation Review, 17(1):39–48, May 1989.

[Cáceres93] Ramón Cáceres, Fred Douglis, Kai Li, and Brian Marsh. *Operating system implications of solid-state mobile computers*. Technical report MITL–TR–56–93. Matsushita Information Technology Laboratory, Princeton, NJ, May 1993.

[Carson92a] Scott Carson and Sanjeev Setia. Optimal write batch size in log-structured file systems. *USENIX Workshop on File Systems* (Ann Arbor, MI), pages 79–91, May 1992.

[Carter91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. *Proceedings of 13th ACM Symposium on Operating Systems*

Principles (Asilomar, CA). Published as Operating Systems Review, 25(5):152-64, 13-16 October 1991.

[CesaBianchi94] N. Cesa-Bianchi, Y. Freund, D. P. Helmbold, and M. Warmuth. *On-line prediction and conversion strategies*. Technical report UCSC-CRL-94-28. Computer and Information Sciences Board, University of California at Santa Cruz, August 1994.

[Chambers90a] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of Self, a dynamically-typed object-oriented language based on prototypes. In Urs Hoelzle, editor, *The Self Papers*. The Self Group, CIS 209, Stanford University, Stanford CA 94305, 1990.

[Comer91] Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP: design, implementation, and internals*, volume II. Prentice-Hall, 1991.

[Cruz92] Rene L. Cruz. Service burstiness and dynamic burstiness measures: a framework. *Journal of High Speed Networks*, 2:105–27. IOS press, Amsterdam, 1992.

[Douglis87] Fred Douglis and John Ousterhout. Process migration in the Sprite operating system. *Proceedings of 7th International Conference on Distributed Computing Systems* (Berlin, 21–25 September, 1987), pages 18–25, R. Popescu-Zeletin, G. Le Lann, and K. H. Kim, editors. IEEE Computer Society Press, 1987.

[Douglis94] Fred Douglis, P. Krishnan, and Brian Marsh. Thwarting the power-hungry disk. *Proceedings of USENIX Winter 1994 Technical Conference* (San Francisco, CA), pages 292–306. USENIX Association, Berkeley, CA, 17–21 January 1994.

[Golding94] Richard Golding, Carl Staelin, Tim Sullivan, and John Wilkes. "Tcl cures 98.3% of all known simulation configuration problems" claims astonished researcher! *Proceedings of Tcl/Tk Workshop, New Orleans, LA*, June 1994. Available as Technical report HPL–CCD–94–11, Concurrent Computing Department, Hewlett-Packard Laboratories, Palo Alto, CA.

[Greenawalt94] Paul M. Greenawalt. Modeling power management for hard disks. 2nd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS '94) (Durham, NC), pages 62–66. IEEE Computer Society Press, 31 January–2 February 1994.

[Herrin91] Eric H. Herrin II and Raphael A. Finkel. An implementation of service rebalancing. Technical report 191–91. University of Kentucky, Department of Math Sciences, July 1991. Proc. of the XI Intl. Conf. of the Chilean Computer Science Society, 1991.

[HPKittyhawk92] Hewlett-Packard Company, Boise, Idaho. *HP Kittyhawk Personal Storage Module: product brief*, Part number 5091–4760E, 1992.

[Jacobson91] David M. Jacobson and John Wilkes. *Disk scheduling algorithms based on rotational position*. Technical report HPL-CSP-91-7. Hewlett-Packard Laboratories, 24 February 1991.

[Karn91] Phil Karn and Craig Partridge. Improving roundtrip time estimates in reliable transport protocols. *ACM Transactions on Computer Systems*, **9**(4):364–73, November 1991.

[Lindsey81] William C. Lindsey and Chak Ming Chie. A survey of digital phase-locked loops. In William C. Lindsey,

editor, *Phase Locked Loops*, pages 296–317. Institute of Electrical and Electronics Engineers, April 1981.

[Litzkow88] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor—a hunter of idle workstations. *Proceedings of 8th International Conference on Distributed Computing Systems* (San Jose, CA), pages 104–11. IEEE Computer Society Press, 13–17 June 1988.

[Marsh93] Brian Marsh, Fred Douglis, and P. Krishnan. Flash memory file caching for mobile computers. Technical report MITL-TR-59-93. Matsushita Information Technology Laboratory, Princeton, NJ, 18 June 1993.

[Massalin89a] Henry Massalin and Calton Pu. Fine-grain scheduling. *Proceedings of Workshop on Experience in Building Distributed and Multiprocessor Systems* (Ft. Lauderdale, FL), pages 91–104. USENIX Association, October 1989.

[McDonald89] M. Shane McDonald and Richard B. Bunt. Improving file system performance by dynamically restructuring disk space. *Proceedings of Phoenix Conference on Computer and Cmm.* (Scottsdale, AZ), pages 264–9. IEEE, 22–24 March 1989.

[McVoy91] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIX file system. *Proceedings of Winter 1991 USENIX* (Dallas, TX), pages 33–43, 21–25 January 1991.

[Postel80a] J. Postel. *Transmission Control Protocol*, Technical report RFC–761. USC Information Sciences Institute, January 1980.

[Rodeheffer91] Thomas L. Rodeheffer and Michael D. Schroeder. Automatic reconfiguration in Autonet. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, CA). Published as *Operating Systems Review*, **25**(5):183–97, 13–16 October 1991.

[Rosenblum92] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, **10**(1):26–52, February 1992.

[Ruemmler91] Chris Ruemmler and John Wilkes. *Disk shuffling*. Technical report HPL–91–156. Hewlett-Packard Laboratories, October 1991.

[Ruemmler93] Chris Ruemmler and John Wilkes. UNIX disk access patterns. *Proceedings of Winter 1993 USENIX* (San Diego, CA), pages 405–20, 25–29 January 1993.

[Ruemmler94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, **27**(3):17–28, March 1994.

[Seltzer90b] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. *Proceedings of Winter 1990 USENIX Conference* (Washington, D.C.), pages 313–23, 22–26 January 1990.

[Seltzer93] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. *Proceedings of Winter 1993 USENIX* (San Diego, CA), pages 307–26, January 1993.

[Shapiro92] Marc Shapiro, Peter Dickman, and David Plainfossé. *SSP chains: robust, distributed references supporting acyclic garbage collection*. Technical report 1799. INRIA, France, November 1992.

[Vongsathorn90] Paul Vongsathorn and Scott D. Carson. A system for adaptive disk rearrangement. *Software—Practice and Experience*, **20**(3):225–42, March 1990.

[Wilkes92b] John Wilkes. *Predictive power conservation*. Technical report HPL—CSP—92–5. Concurrent Systems Project, Hewlett-Packard Laboratories, 14 February 1992.

Author information

Richard Golding received his M.S. degree in 1991 and his Ph.D. degree in 1992, both in Computer and Information Sciences from the University of California, Santa Cruz. Following a stint at the Vrije Universiteit Amsterdam, he joined Hewlett-Packard Laboratories as a researcher. His research interests include distributed operating systems, wide-area networking, and storage systems.

Peter Bosch graduated with degrees in Electrical Engineering (1988), and Computer Science (M.Sc., 1994) from the University of Twente, Netherlands. He has worked since 1991 for the University of Twente as an research assistant. His current work involves research into file systems and implementation of a high performance file system for the ESPRIT Pegasus project. His hobbies are traveling and spending evenings in nice restaurants.

Carl Staelin received his Ph.D. in Computer Science from Princeton University in 1992. He has worked since 1992 as a researcher for Hewlett-Packard Laboratories. As part HP's Berkeley Science Center he is currently working on the NOW and Mariposa projects at U.C. Berkeley. His current research interests include {high-performance, tertiary, distributed, reliable} storage systems, distributed systems, and electronic libraries. For fun he is also working to provide a library of binary packages of public domain software for HP workstations.

Tim Sullivan has been a member of the technical staff at Hewlett-Packard Laboratories since 1984, when he was an undergraduate at Stanford University. He worked on software, workload characterization, and performance modeling tools for experimental I/O devices while obtaining his BS (1985) and MS (1988) degrees in Electrical Engineering. He spent two years at the Hewlett-Packard Laboratories Pisa Science Center in Pisa, Italy designing parallel languages with researchers at the University of Pisa. He has also done research on operating systems and databases for distributed and parallel systems and on high-performance, high-availability storage systems.

John Wilkes graduated with degrees in Physics (BA 1978, MA 1980), and a Diploma (1979) and Ph.D. (1984) in Computer Science from the University of Cambridge. He has worked since 1982 as a project manager and researcher at Hewlett-Packard Laboratories in Palo Alto, CA. His current research area is high-performance, high-availability storage systems, and he has interests in multicomputer interconnects, operating system design, and performance modelling. He also enjoys learning about Renaissance art and architecture and interacting with the academic research community.

The authors can be contacted through Richard Golding: golding@hpl.hp.com, Hewlett-Packard Laboratories, mailstop 1U13, PO Box 10490, Palo Alto, CA 94303–0969.

LIBRARIES

Session Chair: Douglas Orr, University of Utah

NOTES

Libckpt: Transparent Checkpointing under Unix

James S. Plank, Micah Beck, Gerry Kingsley University of Tennessee

> Kai Li Princeton University

Abstract

Checkpointing is a simple technique for rollback recovery: the state of an executing program is periodically saved to a disk file from which it can be recovered after a failure. While recent research has developed a collection of powerful techniques for minimizing the overhead of writing checkpoint files, checkpointing remains unavailable to most application developers. In this paper we describe libckpt, a portable checkpointing tool for Unix that implements all applicable performance optimizations which are reported in the literature. While libckpt can be used in a mode which is almost totally transparent to the programmer, it also supports the incorporation of user directives into the creation of checkpoints. This user-directed checkpointing is an innovation which is unique to our work.

1 Introduction

Consider a programmer who has developed an application which will take a long time to execute, say five days. Two days into the computation, the processor on which the application is running fails. If the programmer has not planned for this event, his only choice is to restart the program and lose two days of work. Upon restarting the program, he still needs five days of continuous failure-free processor time to complete the job.

Libckpt is a checkpointing library designed for such a programmer. To use libckpt, all he must do is change one line of his source code and recompile with the library libckpt.a. No other modifications need to be made. Upon execution, the program will periodically save its execution state to disk (at the default interval: every 10 minutes). Upon a processor failure, the programmer need only restart the program with the command line

flag =recover, and the program will roll back to the most recently checkpointed state. In the example above, at most ten minutes of work will be lost, and three more days of *non*-continuous failure-free processor time will be needed to complete the job.

Libckpt is a tool for transparent checkpointing on uniprocessors running Unix. It implements incremental and copy-on-write checkpointing, two optimizations well-known in the literature [2, 3, 9, 10]. Libckpt is a user-level library and uses only facilities which are commonly available under Unix. Libckpt has been ported to and tested on a variety of architectures and operating systems with no kernel modifications. Source code for libckpt can be obtained at no cost by anonymous FTP from cs.utk.edu:-pub/plank/libckpt.

In this paper, we show the performance gains available in libckpt through transparent incremental and copy-on-write checkpointing. In addition, we introduce a new optimization technique, implemented in libckpt, called user-directed checkpointing. User-directed checkpointing works under the assumption that a little information from the user can yield large improvements in the performance of checkpointing. We demonstrate that this assumption is often valid.

2 Transparent Checkpointing

The goal of checkpointing is to establish a recovery point in the execution of a program, and to save enough state to restore the program to this recovery point in the event of a failure. The most straightforward method for establishing a recovery point under Unix is to suspend execution of the application while the entire contents of a process's memory and registers are written to a file. This is called sequential checkpointing because disk trans-

fers are not interleaved with program execution. Recovery is effected by reloading the executable from its original file, and then reconstructing the memory and register state from the checkpoint file. This is akin to creating a core file, from which a user may recover using the undump utility and execve().

We say that checkpointing is transparent when no changes need to be made to the application program. While transparency is easy to obtain at the kernel level, it is harder to achieve in a user level checkpointing library. All current implementations of checkpointing share this limitation: They operate transparently and correctly so long as the application is well-behaved in a sense we will define in Section 4. Libckpt and all other user-level checkpointers can cause a correct but ill-behaved application to fail or to produce incorrect output upon recovery.

Checkpointing with libckpt is not completely transparent. The name of the initial procedure in C must be changed from main() to ckpt_target(). This enables libckpt to gain control of the program as it starts, check the command line for the =recover flag, read a file called .ckptrc to set checkpointing parameters, and begin checkpointing. In FORTRAN, libckpt is enabled by changing the main PROGRAM module to SUBROUTINE ckpt_target(). No other program modifications are needed.

By default, once libckpt gets control of a program, it generates a timer interrupt every ten minutes, and takes a sequential checkpoint at each interrupt. This and other defaults can be changed by placing appropriate lines in the .ckptrc file. In this section, we describe all options, where appropriate, as they would appear in the .ckptrc file.

Placing the line "checkpointing <on|off>" in the .ckptrc file turns checkpointing on or off. If off, libckpt will take no checkpoints and will not affect the execution of the application. The default is on.

dir < directory> specifies the directory in which checkpoint files are created and found. The default is the current directory.

maxtime < seconds> defines the interval between checkpoints. At the beginning of the program, and after each checkpoint, libckpt calls alarm(seconds) and takes a checkpoint upon catching each ALRM signal. Setting the timer interval to zero turns off all timer-based checkpointing. The default value of maxtime is 600 (10 minutes).

Many optimizations to simple sequential checkpointing have been described in the literature. Libckpt implements all published optimizations that are applicable to general-purpose uniprocessor checkpointing, as well as the new user-directed optimization. In the remainder of this section, we consider each of them in turn.

2.1 Incremental Checkpointing

When a checkpoint is taken, only the portion of the checkpoint that has changed since the previous checkpoint need to be saved. The unchanged portion can be restored from previous checkpoints. Incremental checkpointing [2, 3, 18] uses page protection hardware to identify the unchanged portion of the checkpoint. Saving only the changed portion reduces the size of each checkpoint, and thus the overhead of checkpointing.

incremental <on|off> turns incremental checkpointing on or off. The default is off.

In general, the size of a non-incremental checkpoint grows very slowly over time if at all. Moreover, only the most recent checkpoint file needs to be retained for recovery — older ones may be deleted. In contrast, old checkpoint files cannot be deleted when incremental checkpointing is employed, because the program's data state is spread out over many checkpoint files. The cumulative size of incremental checkpoint files will increase at a steady rate over time, since many updated values may be saved for the same page. In order to place an upper bound on the cumulative size of incremental checkpoint files, it is necessary to coalesce all old checkpoint files into one new file, and then discard the old files. For this purpose, libckpt includes a utility program ckpt_coa, which coalesces a collection of incremental checkpoint files into a single checkpoint file.

maxfiles $\langle n \rangle$ sets the maximum number of incremental checkpoint files to n. After n checkpoint files have been created, libckpt invokes ckpt_coa to coalesce them into one file. If n=1, then no incremental checkpointing can occur. Values of n greater than one allow the user to strike a balance between the time and space overhead of incremental checkpointing. The default is n=1.

Libckpt uses page protection to identify which pages should be included in incremental checkpoints. Specifically, after initialization and after each checkpoint, the mprotect() system call is invoked to set the protection of all pages in the data

space to read-only. When a write occurs to a memory location in a protected page, the SEGV signal is caught by a handler in libckpt. The faulting page has its access protection set to read-write, and the page is marked as dirty. When libckpt takes the next checkpoint, only the dirty pages are included.

2.2 Forked Checkpointing

Incremental checkpointing as described in the previous section is still sequential: Execution of the application program is suspended while the checkpoint file is written out. An alternative is to make a copy of the program's data space and to use an asynchronously executing thread of control to write the checkpoint file. This is called "mainmemory checkpointing" [10], and improves checkpoint overhead if there is enough physical memory to hold the checkpoint, as the saving of the checkpoint to disk is overlapped with the execution of the application.

The Unix fork() primitive provides exactly the mechanism needed to implement main-memory checkpointing [7, 12]. When forked checkpointing is specified, libckpt forks a child process, which creates and writes the checkpoint file while the parent process returns to executing the application. The fork() system call provides the child with a fixed snapshot of the parent's data space and a separate thread of control.

fork <on|off> in the .ckptrc file turns main forked checkpointing on or off. The default is off.

An important improvement to main-memory checkpointing is copy-on-write checkpointing [2, 9, 10]. Here the copy of main memory is taken using copy-on-write [4, 16]. Many implementations of fork() use a copy-on-write mechanism to optimize the copying of the parent's address space [15]. Thus, forked checkpointing corresponds to either main-memory checkpointing or copy-on-write checkpointing, depending on the operating system's implementation of fork().

2.3 Checkpoint Compression

With checkpoint compression, a standard compression algorithm like LZW [17] is used to shrink the size of the checkpoint [8, 13]. While this may be successful at reducing checkpoint size, it only improves the overhead of checkpointing if the speed of compression is faster than the speed of disk writes, and if the checkpoint is significantly compressed. For uniprocessor checkpointing this

is not the case. Compression has only been shown to be effective in parallel systems with disk contention [13]. For this reason, checkpoint compression is not implemented in libckpt.

3 User-Directed Checkpointing

All the optimizations presented so far maintain the transparency of checkpointing through techniques that are not visible to the typical application program: signal handlers, page protection, and the creation of child processes. In this section, we consider a different approach that can improve on the performance of these transparent techniques and can also substitute for them when automatic mechanisms are not available. We call this approach "user-directed checkpointing." We consider two ways in which user-supplied directives can improve the performance of checkpointing: memory exclusion and synchronous checkpointing.

3.1 Memory Exclusion

There are two situations where the values of memory locations can be excluded from a checkpoint file: when the locations are dead and when they are clean. In the case of dead locations, the values in memory will never be read or written, and thus do not need to be saved. In the case of clean locations, the values in memory exist in a previous checkpoint and have not been changed. Thus they need not be saved in the current checkpoint. While the identification of excludable areas of memory can sometimes be automated (as in incremental checkpointing), libckpt also allows the programmer to declare them explicitly.

For example, suppose the user allocates a large temporary array T to make a calculation. When the *lifetime* of the data in array T is over, it will never be referenced again — the next use of array T will overwrite the old values. If a checkpoint is taken outside of the lifetime of array T, then it can be safely excluded from the checkpoint. Any computation proceeding from this point will not need to use the current values stored in array T.

The stack is a run-time mechanism that helps the checkpointer to determine the lifetime of local variables. This is one form of memory exclusion: Only the live portion of the stack is saved. Unfortunately, this does not work for heap variables or for variables which reside in the statically allocated data segment.

The basis of incremental checkpointing is that clean data need not be repeatedly written to disk. In order to implement automatic incremental checkpointing, libckpt monitors page modifications using the mprotect() system call and a handler for the SEGV signal. This approach has a few weaknesses: It can only operate at the page granularity; system calls can fail rather than generating a SEGV signal when asked to write to a protected page; and on some systems mprotect() is not reliable.

In those cases where automatic mechanisms cannot determine all possible memory exclusions, the performance of checkpointing can suffer. For this reason, **libckpt** allows the programmer to manage memory exclusion explicitly through two procedure calls:

exclude_bytes(char *addr, int size, int usage) include_bytes(char *addr, int size)

Exclude_bytes() tells libckpt to exclude the region of memory specified from subsequent checkpoints. It may be called when the user knows that these bytes are not necessary for the correct recovery from the program. Usage is an argument which currently may have one of two values: CKPT_READONLY or CKPT_DEAD. If the former. then exclude_bytes() has been called because the specified memory will not be written to until the user calls include_bytes() on it. Consequently, libckpt includes this memory in the next checkpoint, but excludes it from subsequent checkpoints until the memory is included with include_bytes(). If CKPT_DEAD is specified, then the memory is dead — it will not be read before it is next written. Thus, libckpt excludes this from the next and subsequent checkpoints, until it is is explicitly included with include_bytes().

Include_bytes tells libckpt to include the specified region of memory in the next and subsequent checkpoints. Thus, include_bytes() cancels the effect of calls to exclude_bytes(), although calls to include_bytes() do not have to match calls to exclude_bytes(). By default, libckpt includes all bytes in a process's active stack and data segments that have not been explicitly excluded.

User-directed memory exclusion can dramatically reduce the size of sequential and incremental checkpoint files, but it must be used very carefully. If a live region of memory is mistakenly excluded from a checkpoint, then a subsequent failure and

recovery can cause an otherwise correct application to fail or to generate incorrect results.

3.2 Synchronous Checkpointing

In the previous section we discuss a mechanism for optimizing asynchronous checkpointing by excluding certain areas of memory. This allows the checkpointer to make use of data lifetime information which would not otherwise be available to it. However, the amount of data which can be excluded from the checkpoint is determined by the program's state when the checkpoint is taken. If the stack is large, or the size of excluded memory is small, then memory exclusion will have little effect.

Synchronous checkpointing is a user directive that allows the programmer to specify points in the program where it is most advantageous for checkpointing to occur. These are called "synchronous" checkpoints because they are not initiated by timer interrupts. Synchronous checkpoints should be inserted by the programmer at points where memory exclusion can have the greatest effect.

checkpoint_here() is a procedure call specifying where a synchronous checkpoint can be taken.

Synchronous checkpoints may be placed in program locations that are reached often. Checkpointing too often, however, can lead to poor performance, and in order to avoid this libckpt allows a minimum interval between checkpoints to be specified.

mintime < seconds> specifies the minimum period of time that must pass between checkpoints. The default is zero. If mintime seconds have not passed since the previous checkpoint, then checkpoint_here() calls are ignored.

Synchronous and asynchronous checkpointing techniques can complement one another. If maxtime seconds have passed and no synchronous checkpoint has been taken since the last checkpoint, then an asynchronous checkpoint is still taken. However, the effect of memory exclusion is likely not to be as beneficial as in a synchronous checkpoint. If both the mintime and maxtime parameters are set, then the former specifies the minimum interval between synchronous checkpoints, and the latter specifies an interval after which an asynchronous checkpoint will be taken. Whenever

a checkpoint is taken, both the minimum and maximum interval timers are reset.

If maxtime is zero, then asynchronous checkpoints are disabled. In this case the specification of memory exclusion can be optimized for synchronous checkpoints, because there is no danger of asynchronous checkpoints being taken.

3.3 An Example

There are many examples where user-directed, synchronous checkpointing can yield large performance gains. Consider the program in Figure 1. This is a typical driver program for many kinds of programs that repeat calculations over numerous points in a data set. Figure 2 shows how one can checkpoint this program with synchronous, user-directed checkpointing in libckpt.

```
main()
{
    struct data *D;
    FILE *fi, *fo;

    D = allocate_data_set();
    fi = fopen("input", "r");
    fo = fopen("output", "w");
    while (read_data(fi, D) != -1) {
        perform_calculation(D);
        output_results(fo, D);
    }
}
```

Figure 1: A typical scientific driver program, no checkpointing

Figure 2: A typical scientific driver program with checkpointing

By specifying that the checkpoint must be taken at the checkpoint_here() call, we are able to omit all of the variable D from the synchronous checkpoint. This is because D is initialized anew at each iteration of the program. If D is large, then user-directed checkpointing will be responsible for a significant savings in checkpoint overhead. Note that this will be a vast improvement over incremental checkpointing because the memory locations in D will be dirty at the time of the checkpoint.

Section 6 shows other successful examples of user-directed checkpointing.

4 The Mechanics of Checkpointing and Recovery

The motivation for checkpointing is to reconstruct the recovery point. We therefore begin with an overview of the recovery process before describing the details of checkpointing. Recovery has four parts: process creation, data state restoration, system state restoration, and processor state restoration.

- Process creation is implemented by invoking the checkpointed program with a special command line argument for recovery. This automatically restores the text portion of the process's state and begins execution. Libckpt parses the command line, detects the argument for recovery, and calls the recovery routine.
- 2. The recovery routine performs the rest of the recovery. Data state restoration means reading the checkpoint file to recreate the contents of data memory: This consists of the process's stack and data segments.
- 3. System state restoration means restoring as much of the operating system state as possible to its state at the time of the checkpoint. Much of the operating system state, such as the process ID and parent process ID, is unrestorable. However, most applications that need checkpointing are what we call "well-behaved," and do not rely on such state. Libckpt determines the state of the open file table at each checkpoint, and saves it as part of each checkpoint. Upon recovery, libckpt restores the system so that the state of open files is the same as it was at the time of the checkpoint. No other system state is either saved or restored by libckpt.

Application	Abbreviation	Language	Running Time (mm:ss)	Maximum Checkpoint Size (Mbytes)	Checkpoint Interval (min)
Matrix Multiplication	MAT	C	15:20	4.6	2
Linear Equation Solver	SOLVE	FORTRAN	13:42	4.6	2
Cellular Automata	CELL	C	17:39	8.4	2
Shallow Water Model	WATER	FORTRAN	25:54	13.1	3
Multicommodity Flow	MCNF	FORTRAN	18:38	24.3	6

Table 1: Description of application instances

4. Processor state restoration requires that processor registers, including the program counter and stack pointers be restored to their values when the checkpoint was taken. In libckpt, we use setjmp() to store the processor state in memory. The processor state is restored using longjmp(). Thus the recovery routine never returns, and execution continues as an apparent "second return" from the setjmp() of the checkpointing routine.

Thus the mechanics of checkpointing are straightforward: When taking a checkpoint libckpt saves the processor state using setjmp() and records the state of the open file table. Then the data state, consisting of the program's stack and data segments, is written to disk.

5 Experiments

In this section, we present the results of checkpointing five application programs using libckpt. The applications are long-running FORTRAN and C programs written by scientists to run under Unix. All are typical of programs that can benefit from checkpointing for fault-tolerance.

The experiments were performed on a dedicated Sparcstation 2 running SunOS 4.1.3, and writing to a Hewlett Packard HP6000 disk via NFS. The specific instances of the applications are described in Table 1. We describe the applications below:

- Matrix Multiplication (MAT): This is a straightforward matrix multiplication. Two 615 × 615 matrices are read from disk and multiplied, and the product matrix is written to an output file.
- Linear Equation Solver (SOLVE): This
 is a testing program from LAPACK, a highperformance package of linear-algebra subroutines [1]. This program generates a system

of 750 equations with 750 unknowns, uses LU decomposition to solve the system, and then writes the solution to disk. It repeats this process for seven separate systems of equations.

- Cellular Automata (CELL): This program executes a 2048 × 2048 grid of cellular automata for fifteen generations.
- Shallow Water Model (WATER): This is the program STSWM from the National Center for Atmospheric Research. The program is a shallow water model based on the spectral transform method [5]. The instance used here is "Zonal Flow over a Mountain" from their test suite, modeled at 15-minute intervals for six hours.
- Multicommodity Flow (MCNF): This program solves the multicommodity network flow problem using the simplex method [6]. The instance used here runs on a network of 100 vertices and 50 commodities.

Note that for the purposes of these experiments, input values have been chosen to give running times between thirteen and thirty minutes. Typically, the programs would be set up to run for much longer, thus making them ideal candidates for libckpt.

We present results pertaining to the three important metrics of checkpointing performance:

- Checkpoint time: This is the average duration of a checkpoint, from start to finish.
- Checkpoint overhead: This is amount of time added to the running time of the application as a result of checkpointing. Note that in sequential checkpointing, overhead is equal to the total checkpoint time. In mainmemory and copy-on-write checkpointing, the overhead is smaller than the total checkpoint

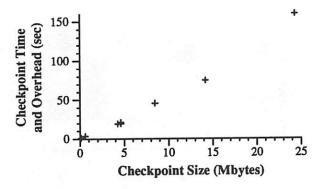


Figure 3: Checkpoint Time vs. Size for Sequential Checkpointing

time because the disk writes are performed in parallel with the execution of the application.

 Checkpoint size: This is the average size of the checkpoint file.

The prime goal of checkpoint optimization is to minimize all three of these metrics, while still providing adequate fault-tolerance. Minimizing checkpoint overhead is the most important, because users would rather take the risk of failure than use a checkpointer that increases their applications' running time significantly. Keeping the overhead of checkpointing under 10% of the program's total running time is a reasonable goal [9, 13]. Minimizing checkpoint size is also important, as disk space rarely comes for free. Checkpoint time is the least important of the three metrics: When checkpointing for fault-tolerance, the only concern is that the current checkpoint complete before the user desires the next checkpoint to begin.

6 Results

All of the experimental results are contained in Table 2 in the appendix. All of the graphs and data in this section are drawn directly from Table 2.

6.1 Sequential Checkpointing

With no optimizations, checkpoint time and overhead should be the same, and should be directly proportional to the checkpoint size. Figure 3 confirms this prediction, showing checkpoint overhead and time vs. size for the sequential checkpointing runs described in Table 2.

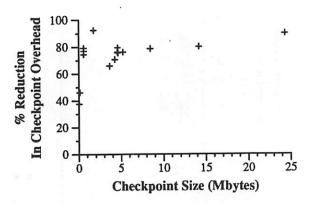


Figure 4: Percentage Reduction in Checkpointing Overhead by Using fork()

6.2 Checkpointing with fork()

When checkpointing with fork(), the application writes its checkpoints to disk asynchronously. This enables it to run concurrently with the saving of the checkpoint, thereby reducing the overhead of checkpointing dramatically, as shown in Figure 4. This figure displays the percentage reduction in the overhead of checkpointing by using fork()¹.

Because SunOS 4.1.3 implements fork() with copy-on-write, Figure 4 shows that copy-on-write improves the overhead of checkpointing by over 70 percent in almost all cases.

6.3 Incremental Checkpointing

Figure 5 is a graph showing the percentage reduction of checkpoint size and checkpoint overhead when using incremental checkpointing instead of simple sequential checkpointing. In three of the applications (MAT, WATER, and MCNF), only a fraction of the applications' address spaces are modified between checkpoints, resulting in a significant reduction in the average checkpoint size. Correspondingly, the overhead of checkpointing is significantly reduced. In the other two programs, the entire address spaces of the programs are modified between checkpoints, yielding little to no reduction in the size of checkpoints. Therefore, in SOLVE and CELL, the overhead of checkpointing is increased due to the fact that the cost of handling page faults is not offset by a savings in the time to write the checkpoint to disk.

¹Ninety percent reduction in checkpoint overhead means that the overhead of checkpointing using fork() is ten percent of the overhead of sequential checkpointing

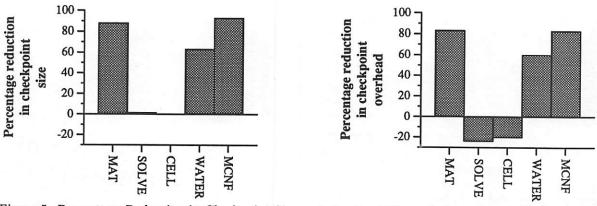


Figure 5: Percentage Reduction in Checkpoint Size and Overhead Through Incremental Checkpointing

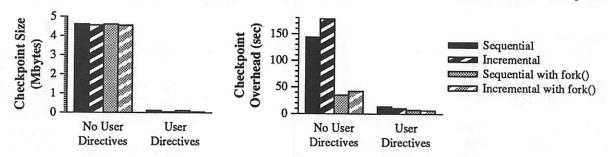


Figure 6: Results of User-Directed Checkpointing on the SOLVE Application

6.4 User-Directed Checkpointing

In the previous three sections, the results corroborate published research concerning check-pointing optimizations [2, 3, 9, 10]. In this section, we evaluate the new technique: user-directed checkpointing. In three of the applications, we analyzed the application programs and inserted directives in the code. In each case, we were able to add under ten lines of code, making checkpoints synchronous, and excluding memory from these checkpoints. We describe the details of each application below.

SOLVE: Adding directives to the Linear Equation Solver was straightforward. At the end of each iteration, all of the program's arrays are dead: The matrix of equations will be initialized anew for the next iteration, and the solution vector will be recalculated. Therefore at the end of each iteration, we insert exclude_bytes() calls for the equation matrix and solution vector, then a checkpoint_here() call, and finally include_bytes() calls to re-include the matrix and vector in case of an asynchronous checkpoint.

The results can be seen in Figure 6: The calls to exclude_bytes() and checkpoint_here() produce checkpoint files that are almost, reducing the checkpoint size and overhead by over 90 percent.

This is significant, because it is an application where incremental checkpointing fails to improve the performance of checkpointing.

CELL: At the end of each generation of the cellular automaton application, the previous value of the automaton grid becomes dead — its values are not used for the calculation of the subsequent generations of the computation. Therefore we added user directives to checkpoint at the end of each generation, excluding the dead half of the grid from each checkpoint. In order to checkpoint at roughly the same interval as before, we also set mintime to 100, so that every second generation is checkpointed.

The results are in Figure 7. With our user directives, the checkpoint size is halved. Accordingly, the overhead of checkpointing is also halved. Thus, as in SOLVE, the calls to exclude_bytes() and checkpoint_here() succeed in improving the overhead of checkpointing in an application where incremental checkpointing fails.

MAT: In the matrix multiplication the two input matrices are read-only data. Moreover, once a product element is calculated it too is read-only. This is why incremental checkpointing works so well. In this application, we inserted exclude_bytes() calls (with flag=CKPT_READONLY) after reading the input ma-

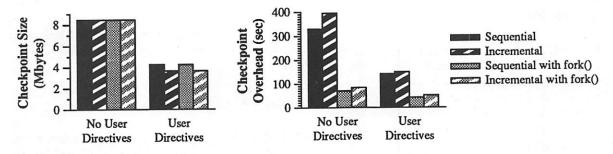


Figure 7: Results of User-Directed Checkpointing on the CELL Application

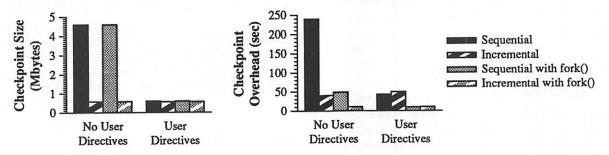


Figure 8: Results of User-Directed Checkpointing on the MAT Application

trices and also after calculating a product row to mark the memory as read-only. Thus, once a checkpoint contains these values, subsequent checkpoints omit them. The behavior of the application with these calls should approximate standard incremental checkpointing — after data becomes read-only, it is omitted from subsequent checkpoints.

The results of this experiment are shown in Figure 8. The important bars are the solid ones, showing that the checkpoints obtained with the user directives are approximately the same size as those obtained with incremental checkpointing. Moreover, they show slightly lower overhead, because they spend no extra time catching page faults.

7 Related Work

There has been much computer science research devoted to checkpointing. Checkpointing has been implemented on uniprocessors [8, 11], multiprocessors [9, 10], transputers [14], multicomputers [13], and and distributed systems [2, 7]. Of these implementations, only two (Condor [11] and Fail-Safe PVM [7]) are publicly available code for Unix environments. Both implement sequential checkpointing with forking, and neither is designed for simple uniprocessor checkpointing: Condor is a system for batch programming using process migra-

tion, and Fail-Safe PVM requires the programmer to have access to the PVM infrastructure. Neither package implements any optimizations beyond calling fork().

User-directed checkpointing bears some similarity to checkpointers by Li and Fuchs [8], and Silva et al [14]. The former describes static checkpointing, which is similar to our synchronous checkpointing. The user places potential_checkpoint_here() calls into his program, and the compiler and/or runtime system decides which of those calls would be best for checkpointing. They call for no user assistance in determining the memory to exclude (they only exclude the stack and unallocated heap memory), and do not show the dramatic performance improvements gained by user-directed checkpointing in the SOLVE, CELL, and MAT applications.

Silva et al implement a checkpointing package for transputers in which the user specifies exactly what and where to checkpoint, but the process state is not included in checkpoints. Thus the user is responsible for rebuilding the call stack, although not the data, on recovery. Our approach differs because the checkpointer is responsible for the entire process state, and not just for the integrity of the data.

8 Conclusion

We have written a general-purpose checkpointing library, libckpt, that provides fault-tolerance for long-running programs under Unix. The strengths of this library are its ease of use and low overhead. Libckpt is currently available via anonymous FTP to cs.utk.edu in the directory pub/plank/libckpt.

Our experiments with libckpt show first and foremost that it is general-purpose and easy to use. We were able to checkpoint all five applications by changing one line of the applications' source code, and relinking with libckpt. Once enabled, these programs could save their state to disk periodically for fault-tolerance, using the fork() and incremental checkpointing optimizations if so desired. For all five applications, we were able to dramatically lower the overhead of checkpointing with copy-on-write, as implemented by libckpt's fork() optimization. Moreover, in three of the five applications, checkpoint size and overhead were reduced by over 60 percent using incremental checkpointing. Thus, libckpt is able to take efficient checkpoints using standard techniques from the checkpointing literature.

Libckpt also implements user-directed checkpointing, a new technique for improving the performance of checkpointing based on the assumption that a little user input to the checkpointer can result in a large performance payoff. Memory exclusion and synchronous checkpointing are the two ways in which a user can direct the checkpointer to checkpoint more efficiently. In our experiments, directives added to three of the applications yielded performance improvements in all three cases.

One avenue of future research is to employ compiler analysis to assist user-directed checkpointing. If the user places the checkpoint_here() calls, the compiler can use data dependence analysis to make calls to exclude_bytes() and include_bytes(). The benefits may be twofold. First, the compiler may discover dead variables to exclude that the user may omit. Second, the compiler can guarantee that its memory exclusion will yield correct checkpoints. In other words, whereas the user might err in excluding too much memory from a checkpoint, resulting in a faulty recovery state, the compiler can guarantee correctness.

It is the authors' opinion that checkpointing primitives such as those provided by libckpt should be implemented in the operating system. This will improve both the performance and the

generality of checkpointing. Until such a time, users can make use of a tool such as libckpt to render their programs resilient to failure.

Acknowledgements

The authors thank Jian Xu, Jack Dongarra, Christian Halloy, and the National Center for Atmospheric Research for help in obtaining test programs. We also thank Nitin Vaidya, Mootaz Elnozahy, Heather Booth, and the referees for their valuable comments. James Plank is supported by NSF grant CCR-9409496. Kai Li is supported by ARPA and ONR under contracts N00014-91-J-4039, and Intel Supercomputer Systems Division.

References

- [1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. SIAM, Philadelphia, PA, 1992.
- [2] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In 11th Symposium on Reliable Distributed Systems, pages 39-47, October 1992.
- [3] S. I. Feldman and C. B. Brown. Igor: A system for program debugging via reversible execution. ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging, 24(1):112-123, Jan 1989.
- [4] R. Fitzgerald and R.F. Rashid. The integration of virtual memory management and interprocess communication in accent. ACM Transactions on Computer Systems, 4(2):147-177, May 1986.
- [5] J. J. Hack, R. Jakob, and D. L. Williamson. Solutions to the shallow water test set using the spectral transform method. Technical Report TN-388-STR, National Center for Atmospheric Research, Boulder, CO, 1993.
- [6] J. Kennington. A primal partitioning code for solving multicommodity flow problems (version 1). Technical Report IEOR-79009, Southern Methodist University, 1979.

- [7] J. León, A. L. Fisher, and P. Steenkiste. Fail-safe PVM: A portable package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, Carnegie Mellon University, February 1993.
- [8] C-C. J. Li and W. K. Fuchs. CATCH -Compiler-assisted techniques for checkpointing. In 20th International Symposium on Fault Tolerant Computing, pages 74-81, 1990.
- [9] K. Li, J. F. Naughton, and J. S. Plank. Realtime, concurrent checkpoint for parallel programs. In Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 79-88, March 1990.
- [10] K. Li, J. F. Naughton, and J. S. Plank. Lowlatency, concurrent checkpointing for parallel programs. *IEEE Transactions on Parallel* and Distributed Systems, 5(8):874-879, August 1994.
- [11] M. Litzkow and M. Solomon. Supporting checkpointing and process migration outside the Unix kernel. In Conference Proceedings, Usenix Winter 1992 Technical Conference, pages 283-290, San Francisco, CA, January 1992.
- [12] D. Z. Pan and M. A. Linton. Supporting reverse execution of parallel programs. ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging, 24(1):124– 129, January 1989.
- [13] J. S. Plank and K. Li. Ickp a consistent checkpointer for multicomputers. *IEEE Parallel & Distributed Technology*, 2(2):62-67, Summer 1994.
- [14] L. M. Silva, B. Veer, and J. G. Silva. Checkpointing SPMD applications on transputer networks. In Scalable High Performance Computing Conference, pages 694-701, Knoxville, TN, May 1994.
- [15] W. Richard Stevens. Advanced Programming in the UNIX Environment. Addison-Wesley, Reading, Mass., 1992.
- [16] M. M. Theimer, K. A. Lantz, and D. R. Cheriton. Preemptable remote execution facilities for the V-system. In Tenth ACM Symposium on Operating System Principles, pages 2-11, Orchas Island Washington, December 1985.

- [17] T. A. Welch. A technique for highperformance data compression. *IEEE Com*puter, 17:8-19, June 1984.
- [18] P. R. Wilson and T. G Moher. Demonic memory for process histories. In SIGPLAN '89 Conference on Programming Language Design and Implementation, pages 330-343, June 1989.

Author Information

Jim Plank is an assistant professor in the Department of Computer Science at the University of Tennessee. He received his Ph.D. degree from Princeton University in 1993. His areas of interest are checkpointing, fault-tolerance, operating systems, and architecture.

Micah Beck is an assistant professor in the Department of Computer Science at the University of Tennessee. He received his Ph.D. degree from Cornell University in 1992. His areas of interest are program analysis and compilation for parallelism.

Gerry Kingsley is a Ph.D. student in the Department of Computer Science at the University of Tennessee. His research interests include compiler analysis techniques, user level checkpointing, and general fault tolerance.

Kai Li received his Ph.D. degree from Yale University in 1986 and is currently an associate professor of the Department of Computer Science, Princeton University. His research interests are in operating systems, computer architecture, fault tolerance, and parallel computing. He is an editor of the IEEE Transactions on Parallel and Distributed Systems, and a member of the editorial board of International Journal of Parallel Programming.

The first three authors' address is: Department of Computer Science, University of Tennessee, 107 Ayres Hall, Knoxville, TN 37996. Their email addresses are [plank, beck, kings-ley]@cs.utk.edu. Kai Li's address is: Princeton University, 35 Olden Street, Princeton, NJ 08544-2087. He may be reached electronically at li@cs.princeton.edu.

Appendix

Appli-	User	Incre-	Fork	Running	Over-	%	Avg Ckpt	Avg Total	Num
cation	Direc-	mental		Time	head	Over-	Time	Ckpt Size	of
	tives			(sec)	(sec)	head	(sec)	(Mbytes)	Ckpts
MAT	No o	checkpoint	ing	920.7	-		-	-	-
	no	no	no	1160.0	239.3	26.0	4.59	21.2	11
	no	yes	no	961.3	40.7	4.4	0.57	3.6	11
	no	no	yes	969.5	48.8	5.3	4.59	22.4	11
	no	yes	yes	931.0	10.3	1.1	0.57	3.4	11
	yes	no	no	963.5	42.8	4.7	0.59	3.6	11
	yes	yes	no	970.5	49.8	5.4	0.56	3.5	11
	yes	no	yes	930.5	9.8	1.1	0.59	5.0	11
	yes	yes	yes	931.0	10.3	1.1	0.56	3.3	11
SOLVE	No c	heckpoint	ing	822.7	-	-	- 12		
	no	no	no	966.5	143.8	17.5	4.62	19.6	7
9 2	no	yes	no	1000.7	178.0	21.6	4.55	24.7	7
-	no	no	yes	857.6	34.9	4.2	4.62	19.6	7
	no	yes	yes	864.8	42.1	5.1	4.55	24.7	7
	yes	no	no	836.3	13.6	1.7	0.11	1.3	7
	yes	yes	no	833.3	10.6	1.3	0.04	1.0	7
	yes	no	yes	830.0	7.3	0.9	0.11	1.4	7
	yes	yes	yes	829.3	6.6	0.8	0.04	1.1	7
CELL	No c	heckpoint	ing	1059.9	-		- 7	-	-
	no	no	no	1389.3	329.4	31.1	8.46	45.4	7
	no	yes	no	1455.4	395.5	37.3	8.44	53.0	7
	no	no	yes	1130.3	70.4	6.6	8.46	43.7	7
	no	yes	yes	1143.9	84.0	7.9	8.44	50.0	7
	yes	no	no	1202.3	142.4	13.4	4.26	19.3	7
	yes	yes	no	1210.3	150.4	14.2	3.63	20.3	7
	yes	no	yes	1101.4	41.5	3.9	4.26	19.3	7
	yes	yes	yes	1111.1	51.2	4.8	3.63	20.4	7
WATER	No c	heckpoint	ing	1553.9	-	-	-	- 1	
	no	no	no	2170.4	616.5	39.7	14.17	74.6	8
	no	yes	no	1800.3	246.4	15.9	5.23	29.9	8
	no	no	yes	1676.1	122.2	7.9	14.17	74.6	8
	no	yes	yes	1612.3	58.4	3.8	5.15	28.5	8
MCNF	No c	heckpointi	ng	1118.2	-30	-			_
	no	no	no	1681.8	563.6	50.4	24.31	159.3	3
	no	yes	no	1216.1	97.9	8.8	1.75	10.7	3
	no	no	yes	1175.2	57.0	5.1	24.31	131.3	3
	no	yes	yes	1125.6	7.4	0.7	1.75	10.3	3

Table 2: Results of all checkpointing experiments

Optimizing the Performance of Dynamically-Linked Programs

W. Wilson Ho Wei-Chau Chang Lilian H. Leung

Silicon Graphics, Inc.

Abstract

Dynamically-linked programs in general do not perform as well as statically-linked programs. This paper identifies three main areas that account for the performance loss. First, symbols are referenced indirectly and thus extra instructions are required. Second, the overhead in run-time symbol resolution is significant. Third, poor locality of functions in shared libraries and data structures maintained by the run-time linker may result in poor memory utilization. This paper presents new optimization techniques we developed that address these three areas and significantly improve the performance of dynamically-linked programs. Also, we provide measurements of the performance improvement achieved. Most importantly, we show that all desirable features of shared libraries can be achieved without sacrificing performance.

1. Introduction

More and more UNIX systems support dynamic linking of shared libraries [MIPS90, Arno86, Ausl90, Cout92, Ging87] because they provide many desirable features. For example, both disk space and physical memory utilization are reduced due to increased sharing, and shared libraries can be replaced transparently without re-linking all user programs. More in-depth discussions can be found in [Ging89, Saba90]. However, the use of dynamic shared libraries does incur a performance penalty. Dynamically-linked programs generally run slower than statically-linked programs because they incur extra run-time overhead. This overhead includes the execution of extra instructions resulted from indirect addressing and run-time symbol resolutions, and extra memory requirement due to poor locality of functions in shared libraries and data structures used by the run-time linker.

Several optimizations have been proposed to improve the performance of shared libraries. Runtime overhead in indirect function calls can be improved by reducing the number of instructions used in the calling sequence [Kepp93]. Symbol resolutions can be deferred and carried out on-demand to improve start-up time [Saba90]. Loading and fixingup of shared libraries can be cached to reduce the amount of work for subsequence invocations [Nels93, Orr93]. These methods are effective in speeding up the execution of dynamically-linked programs from their corresponding initial implementations. However, there still remains significant performance degradation of these programs when compared with their statically-linked counterparts. Informal discussions on the Usenet news group comp.arch seem to suggest that 10-25% degradation is commonly observed.

This paper describes a number of techniques we developed that can bring the performance of dynamically-linked programs much closer to the level of that of statically-linked programs. With these optimizations, we have built the entire IRIX system (SGI's implementation of the UNIX operating system) based on shared libraries without sacrificing performance. In fact, for most libraries, only the shared versions are provided. All optimization techniques described in this paper have been implemented and are available in version 5.3 of the IRIX system.

There are three main areas that account for the performance loss of dynamically-linked programs. First, text and data symbols are referenced indirectly. As a result, more instructions need to be executed. Our approach takes advantage of the information

derived from the cross-module optimization phase of the MIPS compiler [Chow86, Hime87] and converts as many indirect address references as possible to direct references.

Second, symbol resolution takes time, even with deferred binding. Furthermore, since trapping data references is expensive, binding of data symbols is carried out at start-up time. This causes large C++ applications to suffer a long start-up delay because they usually contain a large number of function pointer tables. We developed a scheme called *quick-start* that virtually eliminates any need for run-time symbol binding.

Third, despite the sharing of library text among multiple processes, the use of shared libraries can sometimes require more memory. This extra memory requirement comes from both the data structures used by the run-time linker for symbol resolution for each process and the poor access locality of functions in a shared library. We solved the first problem by precomputing most of the information required by the run-time linker at static-link time and putting them in sharable, read-only segments of shared libraries. To improve the access locality of functions in a shared library, we developed *cord*, a tool that repositions these functions based on either profiling information or explicit user-specification.

The rest of this paper is organized as follows: Section 2 provides an overview of the first implementation of the IRIX shared libraries. Section 3, 4, and 5 present techniques we employed to improve the shared libraries performance in the three respective areas described above. Each section is followed by an analysis of the gains in performance. All measurements are taken on an Iris Indigo, which has a 100 MHz MIPS R4000 processor. Section 6 summarizes the results and offers some conclusions.

2. Background

In this paper, we concentrate on the performance of SVR4-style shared libraries [Syst90], i.e., implementations that support dynamic symbol binding, sharing of text and read-only data among multiple processes, and mapping of the same library to different address space in different processes. In other words, less flexible implementations such as those that require fixed address assignment [Arno86, Hobb87] are not considered.

A full description of the basic implementation of IRIX's shared libraries can be found in [MIPS90]. This section describes only those implementation details that are relevant to the optimizations presented in this paper.

Under the IRIX implementation of shared libraries, the compiler always generates positionindependent code (PIC). Thus, any object file created by the compiler can be part of a shared library. PIC is implemented by turning all address references into indirect references through a Global Offset Table (GOT). A GOT is created at static-link time and there is one for each shared library as well as the main executable itself. It is a table of addresses of all symbols that are referenced, and the content of this table is updated by the run-time linker on demand, i.e., deferred binding. Function calls are implemented by first loading the address of the callee into a register from the corresponding entry in the GOT, followed by a jump-and-link-register instruction. By updating its GOT with correct values, the run-time linker can relocate a shared library to any virtual address.

Furthermore, a dedicated register is used to hold the address pointing to the GOT corresponding to the function being executed. When control is transferred between different shared libraries, the address of the GOT is computed by the corresponding callee and the new value is put in this GOT register. However, when control is passed between functions within the same shared library, the content of the GOT register needs not be changed and can be used immediately.

Using PIC allows the text segment to be shared by all processes because it never needs to be modified. The GOT approach localizes all the run-time fix-up of addresses and cuts down the number of copyon-write memory pages. When compared to the jump table approach in the SunOS compiler [Ging87], our implementation of function calls requires only one control transfer instead of two. This difference is of particular importance in modern computer architecture [Hein93, Hsu94], where jumps are in general expensive. Furthermore, the instruction to load addresses from the GOT can often be scheduled to be executed earlier so that other useful work can be done while the address is being fetched from memory.

3. Eliminating Indirect Addressing

As stated in Section 2, all object files generated by our compiler are PIC. This feature allows any object file created by the compiler to be put into a shared library. It also relieves programmers from the burden of deciding at compile-time whether a particular object should become part of a shared library or an executable.

However, if the main executable is never relocated at run-time, we can assign absolute addresses to it and replace all indirect references for symbols defined in the main executable by direct references. This can be achieved if the system guarantees that no shared library is mapped to the address space already used by the executable itself.

Unfortunately, under the traditional separate compilation model, the compiler cannot determine if references of undefined global symbols will be resolved with a definition from a shared library or from the main executable itself. Hence, it has to assume the worst case and generate indirect references for such symbols.

Our goal is to derive a scheme that can automatically determine if direct addressing is possible for any given symbol, without requiring any specification from the programmer. The approach is to extends the cross-module optimization phase of the compiler to collect information about where symbols are defined.

program	statically	dynamically linked					
	linked	direct a	ddressing	PIC			
espresso	43.2s	45.2s	(4.6%)	47.4s	(9.7%)		
lisp	89.0s	95.9s	(7.8%)	114.8s	(29.0%)		
eqntott	13.9s	13.8s	(-0.7%)	14.0s	(0.7%)		
compress	64.3s	67.0s	(4.2%)	71.8s	(11.7%)		
sc	95.8s	102.7s	(7.2%)	107.3s	(12.0%)		
gcc	120.9s	121.8s	(0.7%)	138.8s	(14.8%)		
average			3.97%		12.98%		

Table 1. Performance Degradation of Dynamically-Linked Programs

Under the cross-module optimization model, the MIPS compiler combines separately-compiled intermediate-code objects and performs function inlining, interprocedural register allocation, dead function removal, and many other optimizations that are not supported in the traditional model [Hime87]. All these optimizations are done before the final machine code generation. Note that during this cross-module optimization phase, the compiler collects information from all the files that compose the program, including the shared libraries that it links with, and thus has complete knowledge of where each symbol is defined. By simply recording this information in the symbol table and passing it down to the code generator, direct references can be generated for symbols defined in the executable module and indirect references can be generated for symbols defined in the shared libraries.

This optimization improves the performance in several ways. First, function call overhead within the main executable is reduced. There is no more memory reference for loading the address of the callee. Also, with the absolute address embedded as an immediate value in the jump-and-link instruction, the processor can potentially pre-fetch the next instruction without delay. Second, overhead for indirect references of data symbols is also eliminated. Very few other implementations, if any, can support dynamically-linked programs with direct data references. This optimization can have a significant effect on performance, especially when the symbols are referenced within a loop. Third, all symbols defined inside the main executable can be removed from the GOT, and thus the amount of work performed by the run-time linker is reduced. Furthermore, since the size of the GOT is reduced, the compiler can put more global variables in the data area close to the GOT and access these variables through a 16-bit immediate offset from the GOT register [Chow87]. Under the MIPS architecture, accessing data via an immediate offset from a register takes only one instruction, and is even faster than the normal direct addressing, which takes two instructions instead of one [Hein93].

Table 1 compares the performance of several dynamically-linked programs against the statically-linked versions. These programs are from the SPECint benchmark suite [SPEC91]. Three versions of each programs were compared: the statically-linked version, the dynamically-linked version with the direct-addressing optimization described in this section, and the default PIC version. The performance of each dynamically-linked version is

compared to that of the statically-linked counterpart. Numbers in parenthesis are the percentage degradation.

By eliminating indirect addressing, the performance of these programs is consistently better than the PIC version. The most significant improvement is found in *lisp* and *gcc*, both of which contains a large number of function calls. When compared with the statically-linked version, the worst performance degradation is cut down to only 7.8%, while *equtott* and *gcc* show almost no degradation. On average, dynamically-linked programs are now only 3.97% slower than statically-linked programs, which is a great improvement over what is commonly observed in other implementations.

Note that only the performance of single-process runs are compared. In general, the use of shared libraries provides a better system throughput because their text pages are shared among different processes, resulting in less memory consumption system-wide. Our results show that such an improved system throughput can be achieved with minimal penalty to the performance of individual process. We discuss the memory requirement of shared libraries in more detail in Section 5.1.

4. Quickstart

One of the major performance degradation in executing dynamically-linked programs is due to run-time symbol resolutions. The performance is affected in two ways. First, more instructions are executed because of the work performed by the run-time linker. Second, if the run-time linker needs to update the GOT's, more copy-on-write pages are needed and thus the residence size of the program is also increased. Since it is too expensive to trap data references, data symbols, including function pointers, are resolved by the run-time linker at program start-up time. As a result, in cases where function pointers are frequently used, as in many C++ programs, deferred binding cannot alleviate the run-time linker's work to a large extent. Furthermore, in some cases, deferred binding of symbols can cause the program to run even slower than immediate binding of all symbols at start-up time. These suggests that minimizing symbol resolution by the run-time linker is crucial in achieving better run-time performance.

A new mechanism called *quickstart* is developed to address this problem. Under quickstart, the

static linker attempts to resolve all symbol references at static link-time, and if none of these references need to be updated at run-time, the run-time linker needs to perform virtually nothing other than the mapping of the required shared libraries. However, to achieve quickstart, the run-time linker must verify that the following two conditions are satisfied before running the user program [Hime90]. First, the shared libraries used at run-time must be the same as those used at static link--time, and second, all libraries are mapped into their pre-assigned default locations and none of the libraries in the list overlaps in virtual addresses.

To assist the run-time linker in verifying the first condition, each shared library is uniquely identified with a name, a time-stamp (the time when it was built), a checksum (the sum of the external interfaces and sizes of all common symbols), and a version string. At static link-time, all the information about the shared libraries used in building a certain binary are recorded in the binary itself, regardless of whether the binary is an executable or another shared library. In addition, symbols that have multiple definitions are recorded in the binary to enable the runtime linker to correctly resolve references to these symbols. To satisfy the second condition, we support a registry file that the static linker uses to determine the default virtual addresses for each individual shared library, which ensures that the shared libraries are assigned non-overlapping addresses.

As soon as the run-time linker detects that any of those two conditions is not met, it has to perform all the necessary symbol resolutions and relocations before running the program. This is known as *non-quickstart*.

Even with quickstart, the run-time linker still needs to perform symbol resolutions on those multiply-defined symbols, if they exist. In order to reduce the amount of work, we implemented *hidden* symbols. Hidden symbols can *only* be referenced within the shared library that they are defined in. Since these symbols cannot be referenced by other binaries, they directly contribute to making the list of multiply-defined symbols smaller, effectively alleviating the work of the run-time linker.

To handle those cases when the quickstart conditions are violated, e.g., when some of the shared libraries used in an executable are replaced, we developed a system that traverses the entire list of installed system binaries, performs necessary symbol

resolutions and relocations, and at the end rewrites those modified binaries in place. All of these are typically done transparently during new software installation. We call this system *re-quickstart*. This scheme turns out to be very successful in maintaining quickstart status without re-linking any executable or shared library.

It should be pointed out that quickstart is *only* an optimization feature, and has no bearing on the correct execution of a program. Without quickstart, the run-time linker performs deferred binding by default, while immediate binding of symbols is also provided as an option.

Table 2 shows the performance improvement of quickstart and deferred binding over immediate binding. For each program, the total time spent in the run-time linker is measured. The percentage improvement is given in parenthesis. These five programs are chosen mainly because of their popularity of use by programmers. The other reason is their differences in characteristics such as the number of shared libraries used and the number of data symbols that needs to be resolved versus that of text symbols. DiskView, a huge X-window application written in C++, is a graphical interface for disk and file system manipulation used frequently by system administrators. Insight, a large multi-media application, is used for searching and browsing through on-line documents. Gdiff, a medium-sized X-window program, is a graphical differential file comparator. Dbx, a small C++ program, is SGI's source-level debugger. Last but not least, ls, listing the contents of a directory, is a widely used UNIX command.

	# of libs.	non-		
program		immed. binding	deferred binding	quickstart
diskview	17	3.66s	3.43s (6%)	2.06s (44%)
insight	14	3.36s	1.59s (53%)	0.13s (96%)
gdiff	6	0.28s	0.31s (-11%)	0.04s (84%)
dbx	3	0.32s	0.32s (0%)	0.02s (94%)
ls	1	0.04s	0.04s (0%)	0.00s (99%)
average			10%	83%

Table 2. Performance Improvement of Quickstart and Deferred Binding over Immediate Binding.

Overall, programs that are quickstarted spend at least 40% less time in the run-time linker than those with immediate binding. On average, there is an impressive 83% improvement. Most programs show that deferred binding of symbols has advantage over immediate binding [Saba90], and it is more apparent in cases where the number of shared libraries used is larger. This is probably because in general most library functions in a shared library are not used and thus need not be resolved at all. In the case of gdiff, however, the run-time linker was invoked many times over the course of execution, owing to many function calls that required symbol resolution, resulting in longer time spent in the runtime linker. In conclusion, whether deferred binding is used or not, it is far better to maintain the quickstart status, requiring the run-time linker to perform the least amount of work. It should be mentioned that even though only five programs are shown here, we have done substantial testing on hundreds of applications and the results are unanimously in favor of quickstart.

5. Reducing Memory Requirements

One of the goals of shared libraries is to reduce the system-wide physical memory requirement through increased sharing of text and read-only data of shared libraries among multiple processes. By reducing the system-wide memory requirement, the number of page faults is generally reduced, thus improving the overall performance.

While the use of shared library does reduce memory requirement, this reduction is sometimes offset by other overhead of dynamic linking. The first problem is that locality of functions within a shared library might be poor. A shared library is created by combining all functions it provides into a single module. When a shared library is linked with a process, the entire module is mapped into the process' address space and every function defined in the library is available to the process. Only those memory pages corresponding to functions that are actually invoked are loaded on demand. Typically, a process uses only a subset of the functions provided by a shared library. Unless these functions are all packed closely together within the shared library, many memory pages that are loaded might be partially filled with functions that are never or rarely invoked. As a result, memory utilization is sub-optimal.

The second problem is that data structures used by the run-time linker for symbol resolutions may sometimes be large. For example, the IRIX implementation of xterm, a typical X application, links with seven shared libraries containing a total of 5856 symbols. Note that these data structures are private to each process and are not shared. The effect of this hidden cost of dynamic linking has rarely been studied.

Section 5.1 describes *cord*, a tool we developed to improve function locality within a shared library. Sections 5.2 presents an optimization to reduce the amount of memory required by the run-time linker.

5.1. Procedure Repositioning

To reduce system-wide memory usage of instruction text space, a post-linking binary optimization tool called cord is implemented to reposition procedures in executables and shared libraries. Cord attempts to reduce run-time memory usage and achieve better instruction cache mapping by grouping frequently used procedures on the same page or adjacent pages, thus reducing the total number of memory pages needed during program execution. The procedure repositioning is controlled by a feedback file generated by profiling tools. The feedback file can also be hand-tuned to directly control the procedure placement.

Cord and the profiling system can be used to reposition procedures based on three different criteria: procedure invocation count, procedure cycle count and procedure density (procedure cycle count divided by procedure size). Using procedure invocation count to guide procedure repositioning improves memory usage by grouping frequently invoked procedures into adjacent pages. Procedure repositioning based on procedure density is intended to give smaller procedures a heavier weight in deciding the order of repositioning. Earlier experiments show that the effects on performance and memory usage of repositioning based on these three factors varies, but show very little difference. Our results indicate that cycle count yields a small advantage in the overall performance. Therefore, it is chosen as the default and is used in the measurement presented in this section.

The fact that libraries may be shared by different processes makes the performance modeling and measurement of shared libraries more difficult. In

contrast, the benefit of procedure repositioning is more apparent for executables, because their execution pattern is more tractable. As a result, the benefit of procedure repositioning for shared libraries is an interesting issue worth investigating.

There has been a number of researches on binary repositioning [Hwu89, McFa89, Pett90], including procedure and basic block repositioning. However, none of those addresses memory usage and shared libraries performance, which is one of the important benefits of using cord.

To study the effect of procedure repositioning on memory usage, several key applications, including Xsgi, 4Dwm and xwsh, are selected as targets for memory usage optimization. Xsgi is an X window system server on IRIX. 4Dwm is the IRIX extended Motif window manager. Xwsh is an IRIX terminal emulator. Four libraries are chosen for this measurement because they tend to use more memory during most X sessions. The libraries are libXm, libX11, libc, and libXt.

The executables are profiled by a typical run. For libraries, the repositioning is based on profile data from 4Dwm and xwsh. The details of the profile run for each library are listed in Table 3. 4Dwm uses libXm, libX11, libc, libXt and others. Libraries used in xwsh include libX11, libc, and others.

The measurement is done after re-booting IRIX followed by a login session with 4Dwm, clock, two copies of xwsh, and a number of other X applications. It is important to note that this experiment is carried out on a loaded system. Many other processes are sharing those four libraries in addition to 4Dwm and xwsh. For example, libX11 and libXt is used by xdm, the X display manager, and libc is used

III manus	profiled with			
library	4Dwm	xwsh		
libXm	X			
libX11	X	X		
libc	X	X		
libXt	X	-		

Table 3. How Libraries are Profiled.

by all dynamically-linked processes.

The improvement of instruction memory usage for "cord'ed" executables is listed in Table 4. The figures show the total amount of memory used by each program. The results show that the instruction memory usage has been reduced by an average of 20.1%. These programs typically have long life spans and are frequently run throughout the entire X session. Therefore, the amount of memory usage reduction is quite significant.

The improvement of instruction memory usage for cord'ed shared libraries is listed in Table 5. Taken into account that those libraries are shared by other processes during the measurement, the average instruction memory reduction of 13.35% is quite impressive.

5.2. Pre-compute Run-time Information

As stated in Section 4, a successfully quickstarted process virtually eliminates any processing by the

executable	original (K bytes)	cord'ed (K bytes)	memory usage reduction
xwsh	383	312	18.5%
Xsgi	466	410	12.0%
4Dwm	308	216	29.8%
average	Terre en		20.1%

Table 4. Total Instruction Memory Usage for Procedure Repositioned Executables.

library	original (K bytes)	cord'ed (K bytes)	memory usag reduction	
libXm	1212	914	24.6%	
libX11	561	503	10.3%	
libc	600	526	12.3%	
libXt	351	329	6.2%	
average			13.35%	

Table 5. Total Instruction Memory Usage for Procedure Repositioned Shared Libraries.

program	No. of Symbols	Size of msym tables (Kbytes	
diskview	22505	87.9	
insight	19468	76.0	
gdiff	5745	22.4	
dbx	5324	20.8	
ls	2567	10.0	

Table 6. Size of Msym Tables.

run-time linker. However, when quickstart fails, the run-time linker needs to perform symbol resolution. Typically, for each unresolved symbol, the run-time linker goes through the symbol table of each shared library and searches for a definition. Defined symbols in each symbol table can be accessed via a hash table, which is pre-computed and placed in the read-only segment of a shared library.

To further reduce symbol search time, the runtime linker caches each hash value it computes and keeps these values in a table called the *msym* table, which has a one-to-one corresponding relationship with the symbol table itself. Hence, the hash value of each symbol needs to be computed only once. Given an undefined symbol, the run-time linker first extracts the corresponding msym table for the hash value. It then goes through the hash tables of the main executable and each shared library, and looks for a definition of the symbol using this hash value. As a result, only simple table look-up is needed and runtime symbol resolution is very efficient.

Table 6 shows the sizes of the msym tables for the same set of programs listed in Table 2. For each program, the total number of symbol table entries from the program and the shared libraries is shown. The msym tables contain a four-byte hash value for each symbol table entry. The table shows that the amount of memory used by the msym tables is very large. Furthermore, the msym tables are private to each process. Therefore, in a typical workstation environment with 50 processes, the msym tables can easily take up several megabytes of memory. For example, the libc on IRIX has 2368 symbols. Since every program links with libc, the size of the msym tables for libc alone for 50 processes is about half a megabyte. A simple X application such as xterm requires 23 kilobytes for the msym tables per process.

Fortunately, the memory requirement of the msym tables can be greatly reduced without sacrificing the speed of symbol resolution. This can be achieved by pre-computing the hash value for each symbol and putting the entire msym table into the shared library when it is built. As a result, the sizes of the shared libraries increase, but the msym tables are shared among all processes and so the system-wide memory requirement is reduced. Furthermore, these tables are only loaded on demand. If a process is quickstarted, none of the symbol tables, hash tables, and msym tables is even touched.

6. Conclusion

This paper identifies three main areas that account for the performance loss of dynamically-linked programs, and presents the optimization techniques that we developed to solve these problems. All optimizations described in this paper have been implemented and the results have been proved to be very positive. In fact, we applied these optimizations to build the entire IRIX system based on shared libraries and no noticeable performance degradation can be observed.

We show that the cross-module optimization can eliminate indirect addressing for data references and function calls, and greatly reduces the run-time overhead of dynamically-linked programs. The quickstart mechanism virtually eliminates all run-time symbol resolution. By repositioning the procedures in a shared library and pre-computing the run-time data structures for symbol resolution, the system-wide memory requirement can be greatly reduced.

Most important of all, we show that the flexibility, usability, and other advantages of shared libraries can be achieved without sacrificing performance.

Acknowledgment

We would like to thank Sun Chan, Fred Chow, Seema Hiranandani, and Raymond Lo for their valuable comments on earlier drafts of this paper.

References

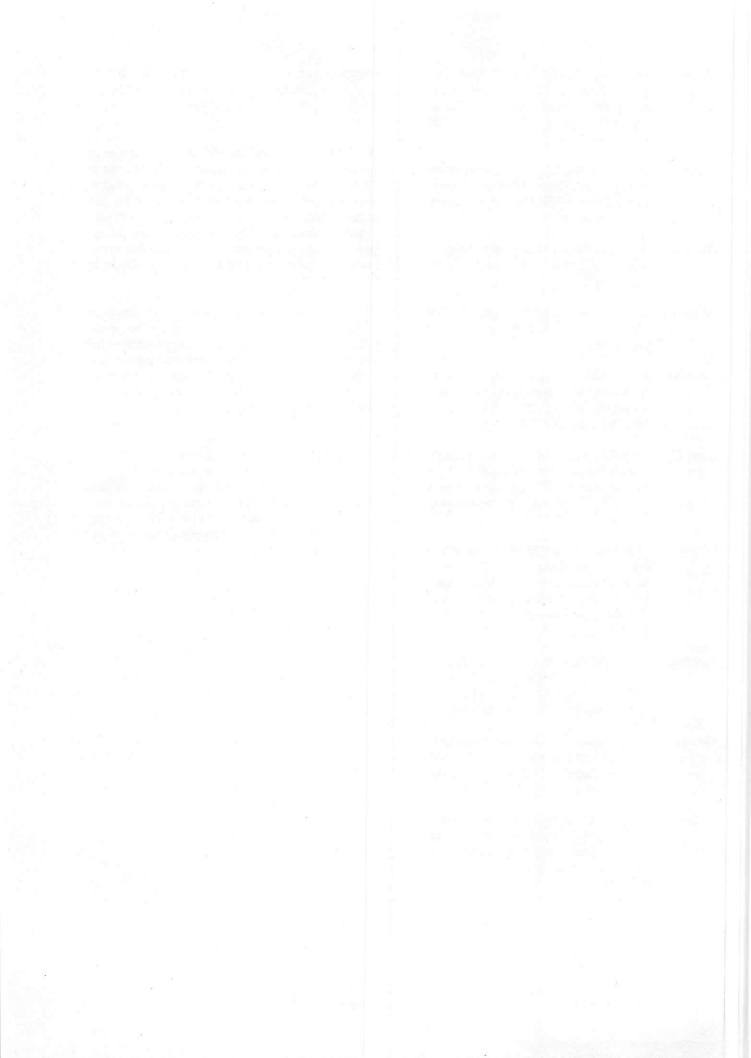
- [MIPS90] MIPS Processor Supplement for the System V Application Binary Interface, Prentice Hall, Englewood Cliffs, NJ (1990).
- [Syst90] System V Application Binary Interface, Prentice Hall, Englewood Cliffs, NJ (1990).
- [SPEC91] SPEC Newsletter. December 1991.
- [Arno86] Arnold, J. Q., "Shared Libraries on UNIX System V," *Proc. Summer Usenix*, pp. 1-10 (1986).
- [Ausl90] Auslander, M. A., "Managing Programs and Libraries in AIX Version 3 for RISC System/6000 processors," *IBM Journal of Research and Development* Vol. 34(1) pp. 98-104 (January 1990).
- [Chow86] Chow, F., M. Himelstein, E. Killian, and L. Weber, "Engineering a RISC Compiler System," *Proc. Compcon.*, pp. 132-137 (March 1986).
- [Chow87] Chow, F., S. Correll, M. Himelstein, E. Killian, and L. Weber, "How Many Addressing Modes are Enough?," Proc. of the 2nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, pp. 117-121 (October 1987).
- [Cout92] Coutant, Cary A. and Michelle A. Ruscetta, "Shared Libraries for HP-UX," Hewlett-Packard Journal, pp. 46-53 (June 1992).
- [Ging87] Gingell, R. A., M. Lee, X. T. Dang, and M. S. Weeks, "Shared Libraries in SunOS," *Proc. Summer Usenix*, pp. 131-145 (Summer 1987).
- [Ging89] Gingell, Robert A., "Shared Libraries,"

 UNIX Review Vol. 7(8) pp. 56-66

 (August 1989).
- [Hein93] Heinrich, Joseph, MIPS R4000 User's Manual, Prentice Hall, Englewood Cliffs, NJ (1993).

- [Hime87] Himelstein, Mark I., Fred C. Chow, and Kevin Enderby, "Cross-Module Optimizations: Its Implementation and Benefits," *Proc. Summer Usenix*, pp. 347-356 (June 1987).
- [Hime90] Himelstein, Mark I., "An Implementation of Dynamic Shared Objects," Unpublished paper, MIPS Computer Systems, Inc., (August 1990).
- [Hobb87] Hobbs, J., "Installed Shareable Images," Dec. Professional Vol. 6(4) pp. 78-82 (April 1987).
- [Hsu94] Hsu, Peter Yan-Tek, "Designing the TFP Microprocessor," *IEEE Micro*, pp. 23-33 (April 1994).
- [Hwu89] Hwu, Wen-mei and Pohua Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler," 16th ISCA, pp. 242-251 (1989).
- [Kepp93] Keppel, David and Stephen Russell, "Faster Dynamic Linking for SPARC V8 and System V.4," Technical Report 93-12-08, University of Washington (1993).
- [McFa89] McFarling, Scott, "Program Optimization for Instruction Caches," International Symposium on Architecture Support for Programming Languages and Operating Systems, pp. 183-191 (April 1989).
- [Nels93] Nelson, Michael. N. and Graham Hamilton, "Higher Performance Dynamic Linking Through Caching," Proc. Summer Usenix, pp. 253-266 (June 1993).
- [Orr93] Orr, Douglas B., John Bonn, Jay Lepreau, and Robert Mecklenburg, "Fast and Flexible Shared Libraries," *Proc. Summer Usenix*, pp. 237-251 (June 1993).
- [Pett90] Pettis, Karl and Robert Hansen, "Profile Guided Code Positioning," Proceedings of the SIGPLAN Programming Language Design and Implementation, pp. 16-27 (June 1990).

- [Saba90] Sabatella, Marc, "Issues in Shared Libraries Design," *Proc. Summer Usenix*, pp. 11-23 (June 1990).
- W. Wilson Ho received his M.S. and Ph.D. degree in computer science from the University of California at Davis. He has been working in the compiler group in Mips, Inc. and Silicon Graphics, Inc. since 1992. His research interests include inter-procedural analysis, dynamic linking, link-time optimization, and debugging of distributed programs. He is also the author the GNU dynamic loader "dld". His email address is ho@sgi.com.
- Wei-Chau Chang has been working in the compiler group in Mips, Inc. and Silicon Graphics, Inc. since 1992. His interests include compiler optimization for instruction cache, program instrumentation, and execution-driven simulation. He received his M.S. degree in computer science from the University of Illinois at Urbana-Champaign.
- Lilian H. Leung received her M.S. degree in Computer Science from Stanford University in 1986. She has been developing compiler at Mips, Inc, and Silicon Graphics, Inc. since 1988. Before that, she was at Informix Software, Inc. designing database languages. Her most recent work focus is on static and run-time linking. She was the original writer of the SGI/Mips' run-time loader.



DP: A library for building portable, reliable distributed applications

David M. Arnow Brooklyn College, CUNY

ABSTRACT

DP is a library of process management and communication tools for writing portable, reliable distributed applications. It provides support for a flexible set of message operations as well as process creation and management. It has been successfully used in developing distributed Monte Carlo, disjunctive programming and integer goal programming codes. It differs from PVM and similar libraries in its support for lightweight, unreliable messages, as well as asynchronous delivery of interrupt-generating messages. In addition, DP supports the development of long-running distributed applications tolerant to the failure or loss of a subset of its processors.

1. Distributed Programming Tools

Although the increase in diversity and availability of parallel multiprocessors shows no sign of abatement, the one truly ubiquitous parallel computer system continues to be the LAN of workstations and the one massively parallel system to which "everyone" has access, if not authorization, is the Internet. Recognition of network computing as an important platform for parallel computing and the desirability of high-level and portable programming systems has resulted in the widespread development of a host of message-passing based programming environments.

Initially, much of this effort went into the design of programming languages or language extensions. An extensive review of these is given by Bal [Bal89]. Each language, besides providing a higher semantic level and portability expresses a view as to how a distributed program ought to be conceived. These views may be limiting. For example, many languages (e.g. SR [Andrews82]) are strongly influenced by the semantic restrictions (synchronous message passing) advocated by Hoare [Hoare78]. Other

languages adopt an asynchronous message semantics (for example, NIL [Strom83]). Still others hide message passing altogether, and present a paradigm different from that of a distributed system. Most notable among these are parallel logic programming languages such as PARLOG [Clark88], the shared memory model of LINDA [Gelernter85], or more recently Concert/C [Aurebach92; Goldberg93].

Languages for distributed systems are necessarily designed with a particular paradigm in mind and as such must impose some restrictions in order to maintain the integrity of that paradigm. The portability of their implementation is not trivial. Perhaps even most significantly, new languages require a substantial reinvestment on the part of users. Therefore, as the locus of interest in parallel network computing changed from language designers to users, there has been a shift to the design of libraries of standard routines or of environments consisting of supporting processes as well as libraries. While the programmer is no longer protected by a language, a greater flexibility and portability can be achieved.

One such environment is PVM [Sunderam90; Geist92], which is implemented on a variety of Unix systems and enjoys extensive use by computational scientists. Others include NMP [Marsland91] and P4 [Butler92]. The performance of these systems and others has recently been reviewed in two papers [Douglas93 and Parsons94]. In response to both the proliferation of such environments and the use of PVM as a de facto standard, the past two years have seen an effort to develop a standard for these environments, MPI [MPI93].

All of these environments provide varying degrees of flexibility, portability and scalability, with PVM providing the most. However, none of them offer the flexibility that my applications required.

Furthermore, none offer the kind of reliability that is necessary for conveniently scaling up to long computations involving many workstations.

2. Wanted: Portability, Flexibility and Reliability

Portability. DP was developed as a result of my own experiences writing distributed programs that ran on LANs and the Internet itself from 1988-1991. These programs included Monte Carlo and other scientific calculations as well as operations research programs. Writing the process management and communication code directly in the native system primitives was maddeningly non-portable even though all the systems involved were either some flavor of Unix or inspired by Unix. The programs were parallelizations of large existing codes, and the necessary interprocess communication was embedded deeply so rewriting these programs in a different language was out of the question.

Flexibility. Most frustrating was the loss of flexibility (with respect to use of the native system primitives) that results from the use of any of the other distributed languages or programming environments available then and now. The communication facilities available in these systems (and described in the proposed MPI standard) do not support *interrupting* messages. Thus, a process receiving a message must invoke a receive operation explicitly at each point that a message is sought. The receive operation is typically allowed to be blocking or non-blocking, so both barriers and polling are readily available to the programmer.

In situations where a process cannot proceed at all until data from an incoming message has been received, these message semantics pose no problem— it is entirely natural to explicitly encode message receive operations just prior to the use of the needed data in the program and entirely proper for these operations to be blocking.

However, there are situations, for example in the Monte Carlo and in disjunctive programming applications in which the I was interested, where:

(a) incoming data serves "merely" to increase the efficiency of the processes computation

and

(b) it is not certain that the incoming data will arrive at all!

In these situations, it is both unnatural and extremely inefficient to explicitly encode non-block-

ing receive operations in the process's application code. In some cases, the problem is mitigated by the availability of threads. A dedicated input thread can integrate the contents of an incoming message into the process's data objects without the need for polling. However, it may be that the only way to efficiently respond to the new information is for the main thread to make an abrupt change in its control, i.e. to make an sudden jump out of its current nested routine stack. In the absence of an inter-thread signaling facility, there is no way for the main thread to recognize the need for this short of testing an object in its own address space— cheaper than executing non-blocking receives, but still inefficient.

Another loss of flexibility is the inability to send fast UDP-style messages in situations where message unreliability may not be a serious drawback. Operating system services often have this characteristic, but, surprisingly perhaps, so do some applications. Consider, for example, a large Monte Carlo calculation involving thousands of random walks. It is often the case that if a small fraction of these are, at random, lost (as a result of message loss), then the impact is "only" an increase in variance. If as a result of permitting such losses, the computation can run faster and hence have a greater sample size, then the benefit could outweigh the loss.

Reliability— for the sake of scalability. Although reliability was decidedly *not* an initial concern of this project, the project's own success forced the issue. The applications that most readily make use of DP are those which have a high computational cost and which are parallelizable. But by running a parallelized application over a great number of workstations on a LAN for a long time, the likelihood of zero workstation reboots during the course of a single computation began to become uncomfortably low. In order to more fully realize the potential for the exploitation of networks and internetworks of workstations, greater reliability is essential. So, as the project developed, increased reliability (with respect to single workstation failure) became a goal.

In summary, the DP library was designed with the following goals:

(1) Flexibility and power: The primitives must provide the power to perform most distributed programming functions. The application programmer should not lose any functionality or efficiency by using DP instead of the native system primitives.

- (2) Portability: The primitives should be implementable on most, if not all, distributed computing platforms.
- (3) Reliability: the loss due to external circumstances of one or all processes on a single workstation should have no impact on the outcome of a distributed computation other than a short delay, provided that the processes involved do not conduct any i/o other than message sending and receiving.

With the exception of not providing a broadcast or multicast message facility, DP meets these goals. On the other hand, in comparison to other distributed programming environments and languages, DP provides a very low-level application interface. There is no typing of messages, minimal support of data format conversions, no queuing of synchronous messages, and no concept of conditional receives. There is, however, a higher-level distributed programming support environments, stdDP [Arnow94] that provides those services and is implemented using DP.

2.1 A sketch of a motivating application: capacitiated warehouse location

To clarify the kind of problem that demands the interrupting message facility that is absent from other environments, we present one example: the capacitated warehouse location problem— a classic operations research problem.

The goal is to supply the needs of customers from a group of proposed warehouses and to minimize monthly cost. Using a warehouse requires a fixed monthly charge and there is the cost of supplying all or part of a customer's requirements from a given warehouse. The problem is to determine which warehouses to open so as to minimize cost while meeting demand. Although the problem is NP-hard, good results can be achieved using Branch-and-Cut and Branch-and-Bound techniques. Worker processes are given portions of the search tree to explore and communicate intermediate results to another. Idle worker processes are given new tasks by master processes which, must obtain these from busy worker processes. Worker processes can complete their tasks significantly more rapidly through pruning by "knowing" the current global minimal cost.

Both of these operations— obtaining the new global minimum and paring the current subtree to define and transmit a new task— are in response to arriving messages, the number of which is unknown.

Polling, besides being inefficient requires an inordinate modification of the original code. In both of these cases, efficiency and convenience is served by providing interrupting messages. This application and other related ones are described in [Arnow91, Arnow94, Arnow95].

3. DP's services

Although of the same genus as environments such as PVM and P4, DP differs from each of them in a number of important ways, primarily because of the above goals. Functionally, the most important difference is its provision for unsolicited messages whose arrival generate a software interrupt. This provides a flexible method of sending large quantities of urgent information that cannot be easily accomplished with, say, the unix-signal transmission of PVM. It also allows a programming environment to provide a shared-memory like capability. DP permits messages to be sent in the cheapest way possible, when reliable transmission is not necessary. Messages can be received with or without blocking. Process creation is dynamic and limited only by available computing resources. Furthermore, DP can guarantee the reliability of those processes that engage in no i/o other than message sending and receiving in the event of a single workstation failure. This section presents an overview of most of the services provided by DP.

3.1 Process management

Execution. Execution of a DP application starts by invoking a suitably compiled and linked DP executable program. In DP parlance, this process is called *the primary*, though its primacy is for the most part just a matter of being first— there is nothing very special about the primary. The primary process and its descendants (those process that are spawned by it or its descendants) constitute a *DP process group*. DP processes can only communicate within this group.

Identification. Each DP process is identified by a value of type DPID, guaranteed to be unique among all possible DP processes. The function dpgetpid() returns the current process's id via a store-back parameter. Processes learn the DPIDs of other processes either by being their parent, by receiving a message from them, or when the DPID of a process is in the contents of a message and is used by the receiving process as such.

The hosts file. In order to spawn processes, a DP program must have information about the available hosts for process creation. DP processes can acquire that information dynamically but it is usually convenient to provide that information to the primary via a hosts file in the directory from which the DP application is executed. The primary will automatically read this file and prompt the user for the passwords needed to access the networks named. This information is inherited by spawned processes.

The hosts table. Host information is maintained internally in a hosts table. In DP application code, hosts are identified using integer indices to this table. The table is dynamic— new hosts can be introduced during run-time by the function <code>dpaddhost()</code>. This call is an alternative and a supplement to providing host information through a static, though convenient, hosts file.

Process creation. Process are spawned by calling dpspawn(), passing

- the name of the program to be executed,
- the integer index of the host on which to run the new process,
- a semantic packet to be sent to the new process; this packet is a program-determined collection of bytes that can be used as an initial parent-to-child communication— typically it contains the DPID of the parent, along with possibly other application parameters.

The call to dpspawn() returns the DPID of the new process via a storeback parameter. The entry point for the newly spawned secondary processes is main(), not the instruction after the call to dpspawn(). These secondary processes do not have access to the original command-line arguments nor do they inherit a copy of the creator's address space—their data must come from the semantic packet or from subsequent received messages. Guiding dpspawn is the entry in the internal host table for the given host id. That entry, among other things, determines the user name under which the new process will run and most significantly the directory in which the program must be found and in which the new process will start executing

Initialization. The first DP call any DP program makes should be dpinit(), which sets up the necessary process and communication environment. This includes initialization of DP's data structures and establishing an address. If the process calling dpinit() was created by dpspawn() the caller is given access to the semantic packet, described above.

A pointer to this packet is returned via storeback parameter to **dpinit()**. The size of the packet is also stored back. In the case of a primary process, there is no semantic packet and the size stored back is -1: that is how the code can determine whether it is running in the primary process or a secondary after a call to **dpinit()**.

The call to <code>dpinit()</code> completes the hand-shaking with the creating parent process. The creating process cannot continue its work until the created process makes this call. For this reason, the call to <code>dpinit()</code> should be made as soon as possible. Upon returning from <code>dpinit()</code>, the process is a genuine DP process and can partake in the activities of the DP family.

In all cases, **dpinit()** returns the number of host machines in its inherited internal host table and stores back the host id of the machine on which the process is running.

Joining a DP process group. Any non-DP process may join an existing DP process group. For this to be possible, one or more of the processes in the group must invoke dpinvite(). This call creates a contact file, which contains all the information that a new process would normally get from dpspawn(). All the joining process need do is invoke dpjoin() with the pathname of the contact file as an argument. This call plays the role of dpinit() and establishes communication using information provided in the contact file. The newly joined DP process's identity can then be conveyed to any process in the group. Note that this mechanism requires that the joining process and the inviting process must share some file address space in common.

Finishing Up. All DP processes must call dpexit() to make a graceful exit. The dpexit() function is the DP substitute for Unix exit() call; that is, it makes a no-return exit. If a DP process fails to exit using dpexit(), i.e. if it exits using the Unix exit(), other DP processes in the application may fail. The main purpose of dpexit() is to withdraw the exiting process from contact with the remaining DP processes prior to an actual exit in a way that guarantees correct message transmission. The only argument to the function is a string identifying the reason for termination. The string appears only in the log file for the process and may be null.

Sometimes, it may be desirable for a process to cease DP activity but persist in some other activity. By passing the address of a function to dpsetex-fun() any time prior to calling dpexit(), a pro-

cess guarantees that dpexit(), after withdrawing from the group of DP processes, will call the indicated function prior to doing the actual Unix exit.

Bailing out. The dpexit() call terminates one DP process in the group. Generally, each process's own logic dictates when that termination is appropriate. In exceptional circumstances, it may be necessary to allow a single process in the group to force termination in the entire group. In such a case, dpstop() can be called. The dpstop() call force immediate shutdown of all processes. The function set by dpsetexfun() is not called and the ensuing shutdown is so radical that even earlier messages that had been sent but were not yet delivered may be thrown away. The function receives one argument, a string, which has the same meaning as the string passed to dpexit().

3.2 Communication

Sending messages. DP processes communicate by sending and receiving messages. For sending messages, the dpwrite() routine requires the DPID of the recipient and a pointer to the start of the message body along with the message body size. A variant, dpsend(), allows a message body to be specified as a linked list. Messages can be reliable or non-reliable and interrupting or non-interrupting. Reliability here means that DP, which as I describe in section 4 uses UDP as its underlying protocol, will carry out an ack/timeout/retransmit protocol that will guarantee the eventual availability of the message to the target provided that the underlying network and relevant host machines do not fail. Reliable messages are received in the order in which they were sent. Sending the message unreliably means that DP will send the message to the target only once and assume no further responsibility— a much cheaper method of message transmission.

Regardless of whether the message is sent reliably, return to the sender is immediate; the sending process will not be blocked during this time. So upon return from **dpwrite()**, one thing is certain: the target has not yet received the message.

Receiving messages. Logically, each DP process has two receiving ports: one for receiving interrupting messages and another for receiving non-interrupting messages. Non-interrupting messages are queued upon arrival and do not affect the receiving process until it explicitly reads the message with the dprecv() call. In the case of the interrupting message, the message's arrival may force the invocation of a special message-catching routine if such a rou-

tine has been designated by the receiving process via a call to dpcatchmsg(). Whether or not such a routine has been designated, the interrupting message must be read explicitly with the dpgetmsg() call, not the dprecv() call. Both routines return the DPID of the sender as well as the message itself and both routines move the incoming message from an internal DP buffer to an application-provided buffer. If the latter is insufficient to hold the message, the message is truncated. The dprecv() call can be made with or without blocking semantics, but the dpgetmsg() call, because it is typically used inside an interrupt handler where blocking would be inappropriate never blocks. In the event that several interrupting messages arrive before the system has had a chance to invoke the message handler function, only one call to the message handler will be made, i.e., there is not a one-to-one correspondence between interrupting messages and calls to the handler. Hence, the message handler must be assume that there may be more than one interrupting message ready to be received.

Longjumps. Sometimes when the message-catching routine is invoked, it responds to the incoming information by modifying a global data structure or sending out a message with requested information. At other times, however, it must respond by making an exceptional change in the control flow of the receiving process. The dplongjmp() routine provides that capability. It works exactly as longjmp() does and in fact its argument is a jmpbuf that was set by setjmp() (there is no "dpsetjmp"). The only reason for dplongjmp() (instead of the standard longjmp()) is that the jump out of the message handler must be accompanied by a re-enabling of interrupting messages.

3.3 Synchronization and timeouts

Critical sections. The application-specified message-catching routine may be invoked at any time and may reference global objects. Thus, any other code that accesses these global objects is a one-way critical section, in the sense that though, upon receipt of an interrupting message control may transfer from the critical section to the handler, the reverse is not possible: control will not pass from the handler until it has completed its work and returns. To guarantee mutual exclusion, such access should be preceded by a call to dpblock() to disable calls to the interrupt handler and followed by a call to dpunblock() to re-enable them. Upon invoking dpunblock(), if any interrupting messages arrived since the call to dpblock(), the catching function will be invoked.

Synchronization and Timeouts. Sometimes a process needs to wait until some condition becomes true, typically as a result of incoming interrupting messages. The dppause() call suspends execution of the process until any asynchronous event takes place. The application may set a timer and a timeout function through non-zero arguments to this call. Upon entering dppause(), interrupting messages (and calls to the message catcher) are enabled and status is restored upon return. Typical use of this function is

dpblock();
while (!some_desired_condition)
 dppause(0, (FUNCPTR) 0)
dpunblock();

The intent of this code is not to proceed until some_desired_condition, which presumably depends on the arrival of a message, is true. Rather than busy-wait, the program calls dppause() which will not return until some event, possibly a message arrival, has taken place. Because many events are possible, the desired condition has to be rechecked and dppause() reentered if necessary. The window between the checking of the desired condition and entry into dppause() open the possibility for a race condition and so the loop is enclosed by calls to dpblock() and dpunblock().

3.4 Restrictions and Application Front Ends

Except for processes that use dpjoin() to join a DP process group, standard input/output/error are not available to the DP application. Thus dpjoin() is essential if interactive programs are desired. Message-catching functions may not call dprecv() in blocking mode.

Timing. All systems calls and standard subroutines that are implemented using the Unix alarm system call (or its variants) are not allowed because they would interfere with DP's own reliance on this facility. That includes: sleep, alarm, ualarm. To restore some of this functionality to the application writer, there is a special DP routine, dpalarm(t,f) which arranges for function f to be invoked after t milliseconds.

Asynchronous and signal-driven i/o. Using the BSD select() system call or making use of the SIGIO signal is forbidden.

Exec and fork. Use of any of the exec variants is forbidden, unless used in conjunction with fork() or after dpexit() has been called. The fork()

system call can be used provided that the children do not attempt to partake in the execution of DP routines. Child processes (but not the parent) may do execs.

Application front ends. These restrictions might initially seem daunting to the application writer. However, it is always possible for non-DP processes, such as one intended to support an event-driven user interface front end, to fork a child process which uses dpjoin() to become a DP process or even which uses dpinit() to become a DP primary process. The non-DP parent and the DP child can communicate using pipes or SysV IPC.

4. Examples

A simple example: primes. The primes program, shown below, illustrates the use of the DP interface. The primary process uses <code>dpcatchmsg()</code> to arrange for <code>fcatch()</code> to be invoked in the event of an interrupting message and then spawns two processes for every available host, sending a semantic packet containing just the DPID of the primary to each secondary process. It divides the interval 1..100000 equally among all the processes, including itself and then uses <code>dpwrite()</code> to send the lower bound of each subinterval to each process in a reliable, non-interrupting message (<code>DPREL|DPRECV</code>). The primary then searches for primes in its own subinterval.

Meanwhile, the secondary processes have started and, having received their lower bound by calling dprecv(), they too start searching for primes in their own subintervals. Both secondary and primary processes invoke newprime() when a prime number is found. For the primary, newprime just adds the prime to the set of primes— this is a critical section because an interruping message may access the same set and so must be protected with dpblock() and dpunblock(). For the secondaries, dpwrite() is used to send the prime number in a reliable, interrupting message (DPREL | DPGET-MSG). The arrival of these message cause fcatch() to be invoked, and the incoming prime number to be stored in the set of primes.

To let the primary know that no more primes are forthcoming, secondaries send a negative integer in a reliable interrupting message and then exit. The primary waits till it has received the appropriate number of such messages and then exits.

```
#include <stdio.h>
#include <dp/dp.h>
struct semstr {
                     /* most programs would have */
                     /* other fields here as well
     DPID s_id;
} s, *sp;
#define MAXPRIMES 100000
int p[MAXPRIMES], np=0, IsPrimary,
     nprocs, nhosts, done=0,
     interval, myhostid;
#defineRelInt(DPREL|DPGETMSG)
#defineRelNonInt (DPREL | DPRECV)
sendint(DPID *dest, int i, int mode) {
     dpwrite(dest, &i, sizeof(i), mode);
}
void
newprime(int n) {
     if (IsPrimary) {
          dpblock(); /* potential race condition */
p[np++] = n;/* so block interrupts */
          dpunblock();
     } else
          sendint(sp->s_id, n, RelInt);
}
void
fcatch() {
     int v;
     DPID src;
     while (dpgetmsg(&src,&v,sizeof(p)
                     !=DPNOMESSAGE)
          if (v<0)
                done++;
          else
                p[np++] = v;
}
search(int n1, int n2)
     int i;
     for (i=n1; i<=n2; i++)
          if (IsPrime(i))
                newprime(i);
}
void
primary(char *prog) {
     int i=1, v=0;
     DPID child;
     FILE *fp;
     dpcatchmsg(fcatch);
     dpgetpid(&s.s_id);
     while (i<nprocs) {
          dpspawn(prog, &child, i%nhosts,
                          &s, sizeof(s));
          sendint(&child, v, RelNonInt);
          1+=interval
          i++:
     }
     search (v, MAXPRIMES);
     done++:
                     /* potential race condition: so */
     dpblock();
     while (done<nprocs) /* block interrupts
```

```
dppause(OL, NULLFUNC);
     dpunblock();
                    /* write primes to results */
     output("results", p, np);
     dpexit("You're fired!");
void
secondary() {
     int v;
     DPID src:
     dprecv(&src, &v, sizeof(v), DPBLOCK);
     search(v, v+interval);
     sendint(sp->s_id, -, RelInt);
     dpexit("I quit!");
               /* main: executed by all processes*/
main(int ac, char *av[]) {
     nhosts = dpinit(av[0], &sp,
                         &size, &myhostid);
     IsPrimary = size==(-1);
     nprocs = 2*nhosts;/* 2 processes per host*/
     interval = MAXPRIMES/(nprocs+1);
     IsPrimary? primary() : secondary();
}
```

Capacitated warehouse location problem, again.

A branch-and-bound search for solutions can be efficently parallelized using DP. N processes are created, N being determined by available hardware. The primary maintains a set of unsearched subtrees-initially this set is take from the top N subtrees of the search tree. When a secondary process becomes idle, it sends an reliable interrupting message to the primary requesting a subtree and then waits until it recieves one. When the primary's set of unsearched subtrees falls below a low-water mark, it sends reliable interrupting messages to all the active secondaries, requesting that they split their subtree at the next convenient point. These will continue to split their subtrees, sending (in reliable interrupting messages) the split-off branches to the primary to replenish its set, until the primary, having passed a high-water mark, sends them reliable interrupting messages to desist. This is very effective load-balancing. The availability of interrupting messages here is essential because of the unpredicatibility of need and availability of search subtrees on the one hand and the undesirability of frequent polling on the other.

An important element of the branch-and-bound search algorithm is the ability to prune search subtrees when the best extremum the subtree can offer is inferior to the best extremum already encountered. In a shared memory environment, all processes have memory access to the best extremum but in a message-passing network environment making sure this information is rapidly available to all processes is both necessary and non-trivial. Using DP, this problem is efficiently addressed as follows. Whenever a

secondary discovers what appears to be a new best extremum it sends a reliable interrupting message to the primary, which multicasts this in lightweight (unreliable) messages to all the other secondaries. Making these message interrupting guarantees that the information will become available to the receiving process as quickly as the underlying system permits. Using lightweight messages ensures that the multicasts will not overload the system nor overbuden the primary. The cost of occasionally losing such a message is minor: it simply means that occasionally for some, usually small, duration, a secondary may not be pruning its subtrees as effectively as it would otherwise.

5. Implementation

DP is implemented using the socket system call interface to the TCP and UDP services of the TCP/IP protocol suite, basic Internet services such as ping and rexec and, of course, a host of Unix services. Once the basic implementation issues were decided all of these services were used in the obvious way.

Communication mechanism. The first issue to be decided was how inter-process communication is to be handled. PVM and many other environments use TCP. This is a very attractive choice given that much inter-process communication has to be reliable and that TCP handles this within the OS kernel. Building a reliable service using UDP requires duplicating much of this outside the kernel with all the context-switching cost that this implies. Nevertheless, DP's inter-process communication is almost entirely implemented using UDP. The reasons for this are:

- TCP requires maintenance of connections and Unix (and presumably most systems) place limits on the number of connections that a process can maintain. The choice then is to take down and recreate connections as needed (too expensive), limit the number of processes with which a process can communicate (clearly unacceptable), or implement a routing mechanism. This would have to be done outside the kernel, negating in part the purpose of using TCP in the first place, especially when large numbers of processes are involved.
- Efficient as TCP is, sending a UDP packet into the ether is cheaper, and because the design of DP was predicated on the desirability of low-cost unreliable messages in many cases, it seemed a shame to pay more than necessary for that kind of communication.

 The fault-tolerant mechanism, described below, is greatly simplified using UDP rather than TCP. This was not a reason at the outset of this project for using UDP, but it became a reason for being very glad about the choice!

Using UDP does require DP to guarantee reliable sequenced delivery of those messages that require this service. In the current implementation, reliable messages are implemented in the most naive way (with sequence numbers, positive acks, timeouts, retransmits and a notion of "stale" messages). More sophisticated implementations are certainly possible and in the still "gray" part of the DP interface, there are calls that allow the programmer to adjust protocol parameters such as timeout size.

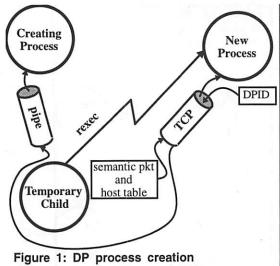
Process identification. It would have been desirable for DPIDs to be integers or some other basic C type. However, that requires some kind of id-to-address mapping internal to DP. A problem arises when an application DP process references a DPID for which its own DP runtime support does not have a mapping. This could and does arise when DPIDs are sent in application messages. To eliminate any need for a centralized or distributed id resolution mechanism, DPIDs are not integers but 28-byte structures containing all the information needed to address the corresponding process and more. The additional information represents "the kitchen sink". Some of it, in retrospect, has turned out to be useful- other components (indicating what protocol- for example IPX— is involved) may never find a use.

There are parallel methods which naturally assign integers to a set of processes and use these assignments in their algorithms. As it turns out, the use of a non-basic type does not pose much of a problem in those cases. Programs which use such methods do not spawn processes dynamically but rather use a fixed number of processes created from the outset. The need to create an initial DPID-to-integer map is only a minor inconvenience to the application writer, and there are libraries, such as stdDP [Arnow, 94] built on top of DP that provide this service, along with others. From an esthetic point of view the chief regret with this choice is the necessity for providing a dpidmatch() function. On the other hand, from an implementation point of view, things are greatly simplified.

Process creation and initialization. Processes are created using the rexec service. In order to spawn a process, the creator, after checking the argument to **dpspawn()** as best as possible, forks a child process which does most of the work. The child process

calls rexec and uses the resulting TCP connection to deliver the semantic packet and the internal hosts table to the newly spawned process. It uses the same connection to receive the new process's DPID (which contains, among other data, its UDP address). This is the only use of TCP in the implementation. The parent receives the DPID from the child through a pipe and waits for the child process to complete the handshaking with the new process and disappear, along with its TCP connection. This arrangement avoids the need for a separate call by the parent to recognize the completion of the process creation. Although it compels the parent to wait until the new process is created, the creator is still able to receive and respond to interrupting messages— an allowance which is made much easier by having a child process do most of the work.

Reliability. The scheme for enhancing reliability is inspired by one used, in a different context, in the early 1980s in the design of a fault-tolerant Unix box based on a shared memory architecture [Borg83]. Each active process is created with a backup process residing on a different workstation. The workstation housing a backup must be binary compatible with that executing the active process. Furthermore, because of the way recovery is implemented, each pair of active and backup processes must share some file address space in common (though between two distinct pairs there is no such need). The scheme only guarantees against single workstation failure, though it may work in the event of multiple failures. What it requires is that the workstation holding the backup process not fail. So in the figure below, had th backup of process C been executing on an additional machine, say Sparc#4, rather than on Sparc#2, then both Sparc#1 and Sparc#2 could have failed simultaneously.



A temporary child process creates the new process using rexec. The resulting TCP connection is used tosend the semantic packet and receive the DPID.

When process A sends a message to process B, process B uses the message and sends a copy of the message to its backup, B'. If the message is a reliable message, B' sends the acknowledgment to A. Thus A continues to retransmit in the usual way until it is certain that both B and B' have the message. Redundant transmits to B cause no problem because they are simply stale messages which an ack/timeout/ retransmit protocol would ignore anyway. B' saves all messages (until, as described below, a checkpoint operation takes place). Upon detection of failure (see below), B' starts executing. Since it has all the messages (with the possible exception of a few unreliables) that B received, its execution will be identical. It will send output messages that are redundant to ones sent by B previously, but these will be treated as stale by their recipients and not cause any inconsistency.

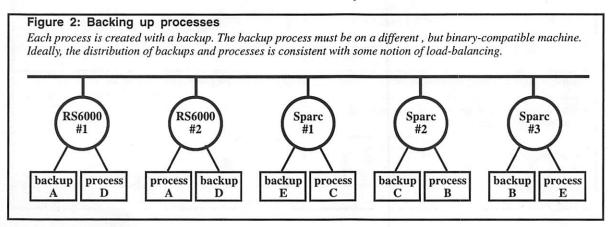


Figure 3.A: Backup Creation

dpspawn() does not return until dpinit() creates the backup, using a modified internal form of dpspawn().

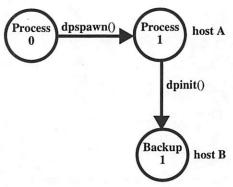


Figure 3.C: Checkpoints

Periodically, a process saves its image in a file accessible to its backup, which is instructed to discard its saved incoming messages.

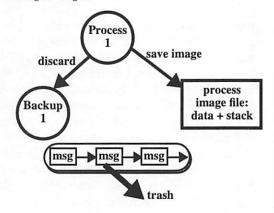


Figure 3.E: Recovery- Phase I

Upon detecting failure, the backup process creates a new backup for itself.

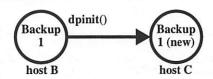


Figure 3.G: Recovery— Phase III

The saved process image file is used to overwrite the backup's data and stack space—the backup now is executing as the original.



Figure 3.B: Saving messages

Copies of reliable messages go to and are saved by the backup. Acks are sent by the backup, not the recipient.

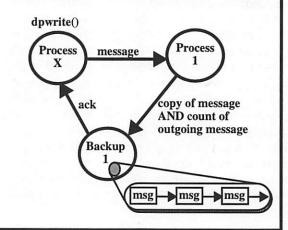


Figure 3.D: Looking for Trouble

The backup process waits in dpinit(), sleeping and pinging the host on which the active process is executing. Timeou on ping means failure.

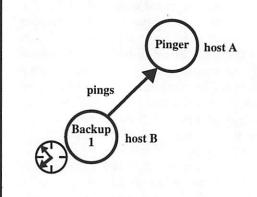


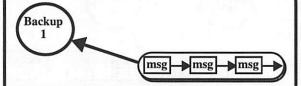
Figure 3.F: Recovery- Phase II

The backup process then informs all other processes that it has taken over for the failed process.



Figure 3.H: Recovery— Phase IV

A period of "roll forward" ensures: the backup reads and processes the save messages.



To preserve the total order of messages as B received them, B assigns each newly received message an internal sequence number. This number is passed on with every copy of that message that B sends to B'.

Failure is detected in an ad hoc fashion. If process B periodically sends dummy messages to its backup, B'. If B' does not hear from B after a time, it pings B's machine. If there is no respond after a number of tries, B' takes over from B.

Recovery begins with B' sending a recovery message to B's parent and every process that B has interacted with. Processes receiving such messages revise their internal DP process tables appropriately and propagate these messages to all other processes they in turn have communicated with. The application code layer never learns about this and will continue to use the original DPID of B which will now be mapped to that of B' by the DP implementation. Delay in this propagation of recovery messages poses no problem because messages sent to B will remain unacknowledged and hence be retransmitted, eventually, to B'. Furthermore, even if B has only failed temporarily (the transceiver cable fell out, say) and comes back to life in the middle of the recovery there still will not be a problem since it always forwards any messages that it receives to B'. Any messages that a temporarily reincarnated B would send will either be stale or cause the equivalent B' message to be treated as such.

Following the transmission of recovery messages by B', is the roll-forward phase. To avoid the delay that would result from having to roll-forward from scratch, active processes periodically (and transparent to the application) have checkpoints, where DP writes out the process's entire stack and data segment to disk. (This is why the active/backup pair must have some file space in common.) At the outset of the roll-forward phase of recovery, these segments are copied into the address space of process B', so the roll-forward starts from the last checkpoint. These checkpoints are done quite infrequently, on the order of 10-15 minutes. The rationale is that anyone running a computation on a group of workstations in which one fails should be grateful to have only a recover delay of 15 minutes. The reason for the fault-tolerance is for computations that run hours, not minutes.

To reduce excess message traffic resulting from redundant messages sent by B' in its roll-forward phase, the active process keeps B' informed as to the number of messages it has sent out to each process. During roll-forward, these counts are decremented and no messages are actually sent out to a given process until its corresponding count has reached 0.

This scheme necessarily requires a number of restrictions on the activities of the active processes. One severe restriction is that they cannot be doing I/O other than DP message transmission (or if they do I/O, its integrity can't be guaranteed).

6. Performance

A number of comparisons of DP's performance with that of PVM have been made. One test involves a ring of processes passing a single message from one to the other. Another involves a set of processes, each of which is sending and receiving messages symmetrically to all the others.

In each test, a DP and an equivalent PVM program were run simultaneously, in the same environment (same machines, same network, same directories, etc.) Each program was given a timeout value and the number of messages passed at that point was measured. Table 1 shows the ratio of messages sent in the DP program to that of the PVM.

TABLE 1. Comparing DP and PVM

Processes	DP/PVM: ring	DP/PVM: set
8	1.33	0.82
16	1.34	0.88
64	1.23	0.81
100	1.36	0.90
112	1.37	1.78

These results suggest DP performs comparably to PVM, that DP may scale better (probably because it does *not* rely on TCP connections) and that a more through performance study is desirable.

7. Portability

DP is implemented on SunOS, Solaris, AIX and on DEC RISCstations. Earlier versions were implemented on the Alliant FX-8 and the KSR-1. A Windows-NT implementation is underway as is a port to NetBSD.

Apart from its reliability enhancement, DP is very undemanding of the underlying system. It requires the ability to spawn remote processes, send messages without blocking and have interrupt-driven input. The reliability enhancement described above has been implemented on SunOS only. Presumably it could be carried out in most Unix environments.

8. Retrospect and Prospects

There seems to be a gap between system designers and application programmers in the area of parallel distributed programming. In this project, I started out wearing an application programmer hat. I had a set of requirements. There was no library then that came close to meeting them and even now no other library meets all of them. At the outset, I had no plan, for example, to provided dynamic process creation. As soon as I wore the system designer hat for a while, that seemed to be a great weakness in the design. It seemed that the flexibility to start with only a few processes and then as new tasks are identified, create additional ones is crucial. Ideally, the programmer would design the process structure of the application to mirror the logical task structure of the problem. After taking the trouble to provide this facility, I was quite chagrined to find that most of the DP users simply assess the number of machines that they have available, choose a number of processes about twice that number, and let them run, using a "worker parallelism" paradigm, in which worker processes are given or pick up tasks as they become idle. The reason for this is understandable. Available hardware is the determining factor in the plans of these practitioners. As I turned to using DP myself, I found I was doing the same thing. Is dynamic process creation really worth the trouble?

On the other hand, the interrupting message facility and the unreliable messages are used extensively. The former in particular has been seen to simplify the parallelization of existing code, by eliminating the need for finding the places in the code to put receives. The interrupting message handler takes care of that. Message sends still need to be inserted into the existing code, but somehow it is easier to identify the points where there is a result to brag about (to other processes say) than to identify receives. In cases where the reverse is true, then sending requests for data can be placed at the appropriate points and the sending of results can be interrupt driven.

On the other hand, the interrupting message facility and the unreliable messages are used extensively. The former in particular has been seen to simplify the parallelization of existing code, by eliminating the need for finding the places in the code to put receives. The interrupting message handler takes care of that. Message sends still need to be

inserted into the existing code, but somehow it is easier to identify the points where there is a result to brag about (to other processes say) than to identify receives. In cases where the *reverse* is true, then requests for data can be placed at the appropriate points (send request in an interrupting message, wait for response) and the actual *sending* of results can be driven by the arrival of these interrupting messages.

The implementation of single processor fault-tolerance invites an effort to undertake process migration and load balancing. Whether the admittedly heavy-handed fault-tolerant scheme used here is efficient enough for that remains to be seen.

9. Availability

DP runs on Sun SPARCstations, DEC RISCstations and on IBM RS/6000s with C and Fortran interfaces. It is, along with documentation and some utilities, available from the author.

10. Acknowledgments

While an undergraduate at Harvard, Haibin Jiu spent two of his summers assisting in this effort. Jim Basney, a student at Oberlin spent a "winter term" on this as well. I especially would like to acknowledge the work of Jerry Chen, who while working on his doctorate at CUNY implemented an early version of DP on the KSR-1 and with whom I have had many valuable conversations.

11. References

- Andrews, G.R.: The distributed programming language SR-- mechanisms, design and implementation. *Software—Practice and Experience 12*,8 (Aug. 1982).
- Arnow, D.M.: Correlated Random Walks in Distributed Monte Carlo Programs. *ICIAM 91*, Washington D.C. (July 1991).
- Arnow, D.M.: StdDP— a layered approach to distributed programming libraries. *T.R.* 94-11 Dept. of CIS, Brooklyn College (1994).
- Arnow, D.M., McAloon, K.M. and Tretkoff, C.: Distributed programming and disjunctive programming. Proceedings of the Sixth IASTED-ISMM Int. Conf. on Parallel And Distributed Computing And Systems Washington D.C. (October 1994).
- Arnow, D.M, McAloon, K.M., and Tretkoff, C.: Parallel integer goal programing. *To appear in the 23rd ACM Computer Science Conference, Nashville.* (1994).
- Aurebach, J., Kennedy, M., Russell, J., and Yemeni, S.: Interprocess communication in Concert/C. T.R. RC 17341, IBM Watson Research Center, Yorktown Heights, (1992).

- Bal, H. E., Steiner, J. G., and Tanenbaum, A.S: Programming languages for distributed computing systems. *Computing Surveys* 21,3 (Sept. 1989).
- Borg, A., Baumbach, J., and Glazer S.: A message system supporting fault tolerance. 9th ACM Symp. on Operating Systems Principles. Bretton Woods, New Hampshire, (Oct. 1983).
- Butler, R., and Lusk, E.: User's guide to the P4 programming system. *Tech. Rep. ANL-92/17*, Argonne Nat. Lab. (1992).
- Chen, J.: Distributed Green's function Monte Carlo calculations. Ph.D Thesis, Dept. of CS, CUNY (1994).
- Clark, K.L.: PARLOG and its applications. *IEEE Transactions on Software Engineering SE-14*, 12 (Dec. 1988).
- Douglas, Craig C., Mattson, Timothy G., and Schultz, Martin H.: Parallel programming systems for workstation clusters. Yale University Dept of CS Technical Report, (Aug., 1993).
- Geist, G.A. and Sunderam, V.S.: PVM— Network-based concurrent computing on the PVM system. *Concurrency: Practice and Experience* 4(4) (Jun., 1992).
- Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems 7, 1 (Jan. 1985).
- Goldberg, Arthur P.: Concert/C Tutorial: An Introduction to a Language for Distributed C Programming, IBM Watson Research Center, Yorktown Heights, (Mar., 1993).
- Hoare, C.A.R.: Communicating sequential processes. *Communications of the ACM 21*,8 (Aug. 1978).
- Marsland, T.A., Breitkreutz, T., and Sutphen, S.: A network multi-processor for experiments in parallelism. *Concurrency: Practice and Experience*, 3(1), (1991).
- MPI Forum: Message Passing Interface Standard (Draft). Oak Ridge National Laboratory. (Nov. 1993).
- Parsons, I.: Evaluation of distributed communication systems. Proceedings of CASCON '93, Vol 2. Toronto, Ontario, Canada (Oct. 1993)
- Strom, R.E. and Yemeni, S.: NIL: An integrated language and system for distributed programming. *SIGPLAN Notes 21*, 10 (Oct. 1983).
- Sunderam, V.S.: PVM— A framework for parallel distributed computing. *Concurrency: Practice and Experience* 2 (1990).

12. Author Information

David Arnow is an Associate Professor in the Department of Computer and Information Science at Brooklyn College. He received an A.B. in Chemistry from Oberlin College in 1973 and a Ph.D. in Computer Science from NYU in 1981. His interests, as one might gather, include parallelizing scientific and operations research programs in a distributed environment. He is also deeply concerned with teaching

Computer Science to undergraduates, regardless of its cruelty. His address is arnow@sci.brooklyn.cuny.edu.

13. Appendix: The DP interface header:

```
/* SEND FLAGS*/
                       0x00 /* not interrupting */
#define
           DPRECV
                       0x01 /* interrupting
           DPGETMSG
#define
                       0x00 /* guaranteed delivery */
#define
           DPREL
                       0x02 /* no guarantee
           DPUNREL
#define
/* RECV MODES*/
                       0x00 /* Wait for message */
#define
           DPBLOCK
           DPNOBLOCK 0x01 /* Don't wait
#define
                             /* RETURN CODES*/
           DPSUCCESS 0
#define
                       (-1)
#define
           DPFAIL
#define
           DPNOMESSAGE (-2)
#define
           DPIDSIZE 28
typedef struct {
      char dpid_info[DPIDSIZE];
      } DPID;
typedef void
                  (*FUNCPTR)();
#define NULLFUNC((FUNCPTR) 0)
int dpinit(char *prog, char *semanticp,
    int *size, *hostid);
      /* RETURNS: DPFAIL or # of available hosts
int dpaddhost(char *hstn, *dmnn, *path,
      *user, *passwd);
      /* RETURNS: number of hosts in host table
void dpgethost(int hid, DPHOST *hptr);
      /* STORES BACK: host info for host #hid
int dpwrite(DPID *dest, char *data,
int nbytes, mode);
/* RETURNS: DPSUCESS or DPFAIL or DPDESTDEAD*/
int dprecv(DPID *src, char *data,
      int limit, int flags);
/* RETURNS: DPSUCCESS or DPFAIL or DPNOMESSAGE*/
int dpgetmsg(DPID *src, char *data,
      int limit);
/* RETURNS: DPSUCCESS or DPFAIL or DPNOMESSAGE*/
int sendflag);
/* STORES BACK the id of the new process and returns DPSUCCESS or DPFAIL
FUNCPTR dpcatchmsg(FUNCPTR f);
/* RETURNS NULLFUNC or ptr to previous catch function*/
void dpexit(char *exitstrng);
      /* removes process from DP group and exits*/
void dpgetpid(DPID *myid);
      /* STORES BACK: DPID of executing process
void dpalarm(long t, FUNCPTR f);
      /* set alarm for user's function f
void dplongjmp(jmpbuf label, int rv);
      /* longjmp to label, return value rv
void dppause(long t, FUNCPTR f);
      /* set alarm for user's function f and pause*/
int dpblock(); /* disable interrupts
void dpunblock(); /* enable interrupts
                                         */
void dpsetexfun(FUNCPTR f)
                                         */
      /* set a f to be called when exiting
void dpstop(char *stopmsg);
      /* abandon ship fast
int dpidmatch(DPID *ip1, DPID *ip2);
      /* return true if match; else false
```

NOTES

FILE SYSTEMS

Session Chair: Noemi Paciorek, Horizon Research

NOTES

File System Logging Versus Clustering: A Performance Comparison

Margo Seltzer, Keith A. Smith Harvard University

Hari Balakrishnan, Jacqueline Chang, Sara McMains, Venkata Padmanabhan University of California, Berkeley

Abstract

The Log-structured File System (LFS), introduced in 1991 [8], has received much attention for its potential order-of-magnitude improvement in file system performance. Early research results [9] showed that small file performance could scale with processor speed and that cleaning costs could be kept low, allowing LFS to write at an effective bandwidth of 62 to 83% of the maximum. Later work showed that the presence of synchronous disk operations could degrade performance by as much as 62% and that cleaning overhead could become prohibitive in transaction processing workloads, performance by as much as 40% [10]. The same work showed that the addition of clustered reads and writes in the Berkeley Fast File System [6] (FFS) made it competitive with LFS in large-file handling and software development environments as approximated by the Andrew benchmark [4].

These seemingly inconsistent results have caused confusion in the file system research community. This paper presents a detailed performance comparison of the 4.4BSD Log-structured File System and the 4.4BSD Fast File System. Ignoring cleaner overhead, our results show that the order-of-magnitude improvement in performance claimed for LFS applies only to meta-data intensive activities, specifically the creation of files one-kilobyte or less and deletion of files 64 kilobytes or less.

For small files, both systems provide comparable read performance, but LFS offers superior performance on writes. For large files (one megabyte and larger), the performance of the two file systems is comparable. When FFS is tuned for writing, its large-file write performance is approximately 15% better than LFS, but its read performance is 25% worse. When FFS is

optimized for reading, its large-file read and write performance is comparable to LFS.

Both LFS and FFS can suffer performance degradation, due to cleaning and disk fragmentation respectively. We find that active FFS file systems function at approximately 85-95% of their maximum performance after two to three years. We examine LFS cleaner performance in a transaction processing environment and find that cleaner overhead reduces LFS performance by more than 33% when the disk is 50% full.

1 Introduction

The challenge in building high performance file systems is in using the disk system efficiently. Since large caches reduce read traffic but do little to reduce write traffic, the critical issue is write performance. Achieving efficient writes to disk implies writing data in large, contiguous units. The central idea in log-structured file systems is that, by aggressively caching data and applying database logging techniques, all disk writes can be made sequential.

The early work in log-structured file systems focused on how to build such file systems. The key issues were providing efficient reads in a file system that was written sequentially and maintaining large contiguous regions on disk. The seminal work on log-structured file systems [9] showed how conventional file system structures could be implemented in an LFS and how the combination of a segmented log and a cleaner process (garbage collector) could be used to maintain large, contiguous regions of disk space. The work's main focus was on the design of log-structured file systems and on the derivation of efficient algorithms for segment cleaning. The performance results reported long-term cleaning summaries (e.g. number of segments cleaned and average utilization

of cleaned segments) and micro-benchmarks that demonstrated the strengths of LFS.

The paper by Seltzer et al. [10] discussed design modifications necessary to incorporate LFS into a BSD framework. The performance analysis presented there focused on areas not covered by Rosenblum and Ousterhout, with an emphasis on workloads that stressed the cleaning capabilities of LFS. It concluded that the clustering modifications to FFS made it competitive with LFS in reading and writing large files and in software development environments (as characterized by the Andrew benchmark), cleaning overhead in LFS degraded transaction processing performance by as much as 40%, and the general applicability of LFS and its competitiveness with FFS warranted further investigation. This paper is part of that further investigation, analyzing the performance of LFS and FFS, and focusing on the areas that pose the greatest challenges to each system. We focus on four main issues:

- validating the BSD-LFS implementation by comparing its performance to that of Sprite-LFS,
- the interaction of file size and performance for sequential access,
- the impact of disk fullness on cleaning overhead in a transaction processing workload, and
- the impact of free space fragmentation on FFS performance.

In Section 2 we compare the BSD implementation of LFS to the Sprite implementation of LFS to validate that we are measuring a representative implementation of a log-structured file system. In Section 3 we examine performance as a function of file size. In Section 4 we examine the performance of the two file systems in a transaction processing environment, with special attention given to LFS cleaning and its performance as a function of disk fullness. In Section 5 we discuss the effect of disk fragmentation on FFS performance. Section 6 summarizes this study.

1.1 Overview of FFS

The BSD Fast File System can be described in terms of its *bitmap*, which keeps track of the free space, and its *cylinder groups*, which correspond to collections of cylinders and provide for regions of allocation and clustering. Information is arranged on disk in terms of three units: blocks, partial blocks called *fragments*, and contiguous ranges of blocks called *clusters*. In principle, placing related files and their inodes in the same cylinder group provides for a high degree of

locality, and allocating blocks contiguously in clusters provides the opportunity for large, sequential reads and writes. In practice, there are two potential problems with FFS. First, operations that affect the file system meta-data (e.g. creating and deleting files) require a large number of I/O operations, many of which are synchronous. For example, it takes potentially six distinct I/O operations to create a new onekilobyte file (the inode of the new file, the directory containing the new file, the directory's inode, the data of the new file, the inode bitmap, and the block bitmap), the first two of which are synchronous. The second potential problem is that the FFS block allocator does not take any special measures to preserve large free extents. File systems that have been in use for a long time may become sufficiently fragmented that it is impossible to find large clusters of blocks. The challenges facing FFS can be summarized as reducing the number and synchronous behavior of writes and avoiding file system fragmentation.

1.2 Overview of LFS

In contrast to FFS, LFS avoids both the multiple-write and synchronous write problems by batching large numbers of writes into large, contiguous writes. However, it must maintain large contiguous regions of free space on disk, called segments. LFS uses a generational garbage collector [5], called the cleaner, to regenerate large free extents. If there is available disk space, the cleaner can always coalesce that free space to produce clean segments. The cleaner can be run during idle periods so as not to interfere with normal file access; however, during periods of high activity it may also be necessary to run the cleaner concurrently with normal file accesses. Depending on the file access patterns, cleaning can potentially be very expensive and can degrade system performance. Thus for LFS the key issue is the cost of cleaning.

2 Validation of 4.4BSD-LFS

The system under study in this work is the 4.4BSD-Lite operating system with implementations of the fast file system (FFS) and the BSD log-structured file system (BSD-LFS), both of which have been improved since the study by Seltzer [10]. Specifically, the read-ahead code used by both FFS and LFS has been largely rewritten. Fragments have been added to LFS. The algorithm for writing blocks to disk in LFS has been modified so that files spanning multiple segments are written with lower-numbered blocks written first. (On a new file system, if N is the number of blocks per segment, blocks zero through N-1 are written to the first segment, N through 2N-1 to the

next segment, and so on.) Finally, the LFS file writing algorithm has been modified so that files are written to segments in the order in which they entered the cache. This does not affect write performance, but improves read performance when files are read in their creation order.

To show that BSD-LFS is a faithful implementation of a log-structured file system, we have run the benchmarks described by Rosenblum and Ousterhout [9] and compared our results to those reported for Sprite-LFS. The two hardware configurations are shown in Table 1. In each benchmark presented in this section, we scale the Sprite measurements so that the performance of the critical components match those of the BSD configuration. Table 2 shows the relevant parameters and their scale factors.

In the following discussion, we refer to the two file systems Rosenblum studied: the Sprite Log-Structured File System (Sprite-LFS) and the default Sun file system without clustering in effect (Sun-FFS) [7], and the three file systems we have studied: the BSD Log-Structured File System (BSD-LFS), the BSD Fast File System with *maxcontig* of one so that clustering is not in effect (BSD-FFS-m1r2), and the BSD Fast File System with *maxcontig* of eight (BSD-FFS-m8r2). As in Rosenblum's tests, we use a 4 KB file system for the LFS implementation and an 8 KB file system for the FFS implementations. We deduced that the *rotdelay* parameter in the Sun-FFS file system

Benchmark Configurations		
	BSD	Sprite
CPU Parameters		
CPU	SpareStation II	Sun 4/260
Mhz	40	25 Mhz
SPEC int92	21.8	8.7
Disk Parameters		
Disk Type	DSP 3105	Wren IV
RPM	5400	3600
Sector Size	512 bytes	512 bytes
Sectors per Track	57	45
Cylinders	2568	1546
Tracks per Cylinder	14	9
Track Buffer	256 KB	32 KB
Average Seek	9.5 ms	17.5
Maximum Bus Bandwidth	2.3 MB/sec	1.3 MB/sec

Table 1: Benchmark configuration. The Sprite column describes the benchmark configuration used in the Rosenblum and Ousterhout study.

Parameter	Sprite	BSD	Scale
CPU (SPECint92)	8.7	21.8	2.5
Disk Bandwidth	1.4	2.5	1.8
Avg Access (I/Os per second)	39	67	1.7

Table 2: Scale factors for Sprite/BSD comparison. In order to validate the 4.4BSD implementation of LFS, we use these scale factors to compare the measurements on different systems. The average accesses per second is based on an average seek plus one-half rotation.

was set to one disk block, indicating that the system was optimized for writing. Similarly, we have optimized the BSD-FFS file systems for writing, setting rotdelay equal to two disk blocks. The tests in Section 3 will examine the interaction of performance and *rotdelay* in more detail.

Rosenblum used two micro-benchmarks to evaluate the performance of Sprite-LFS. The first of these tests measures small-file performance, specifically, meta-data operations (creating and deleting files) and reading small files. Ten thousand one-kilobyte files are created, read in creation order, and deleted. Figure 1 shows the results of this benchmark for the five file systems.

According to Rosenblum, the create and delete tests for LFS are limited by processor speed, while Sun-FFS is limited by disk bandwidth. Our measurements for disk and CPU utilization show that LFS uses 64% of the CPU for creates and 73% for deletes. FFS uses 5-6% for both. Therefore, the LFS results for these tests are scaled by 2.5 (the CPU scale factor) and the Sun-FFS results are scaled by 1.7 (the average access scale factor).

Both log-structured file systems provide an order of magnitude performance improvement over the Fast File Systems for the create phase of the benchmark. Since the files are small, this aspect of the benchmark is dominated by the performance of meta-data operations, and LFS provides superior performance because it batches such operations.

The read test performance is limited by the disk bandwidth, so we scale both Sprite-LFS and Sun-FFS by the bandwidth scale factor (1.8). In both the Sprite benchmarks and the BSD benchmarks, the read performance of LFS and FFS are comparable. The BSD-LFS file system outperforms Sprite-LFS due to larger track buffers on the BSD disk. Once a block of inodes has been read from disk, all the files described by those inodes are resident in the track buffer. Since Sprite's track buffers are one-eighth the size of ours, they cannot cache all the file data.

In the delete test, both LFS systems outperform the FFS systems by an order of magnitude, again due to the meta-data updates, but BSD-LFS outperforms Sprite-LFS by a factor of two. The Sprite-LFS delete performance is baffling. Delete processing in LFS requires the following four steps.

- 1. Remove the directory entry.
- 2. Read the inode.
- Update the appropriate segment usage information to reflect that one kilobyte is no longer in use.
- 4. Increment the version number in the inode map.

The only disk I/O required is reading the directories and the file inode blocks. As in the create case, we expect this test to be limited by the CPU. However, the CPU processing required for deletes is greater than that required for creates, and the amount of data written for deletes is less than that required for creates. The Sprite-LFS results indicate that the CPU was saturated during the create test, so we cannot explain why the Sprite-LFS delete performance exceeds the create performance. We expect the deletes to occur at close to memory speed with a single write at the end of the benchmark to write the empty directories, the inode map, and segment usage table. This is consistent with the BSD-LFS results.

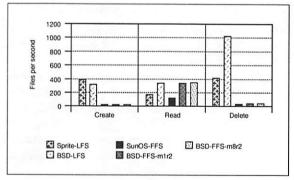


Figure 1. Validation of the BSD-LFS small file performance. The Sprite-LFS results have been scaled to compensate for the performance of the different processors and disks on the two systems. The LFS create and delete tests are scaled by the CPU scale factor, the FFS create and delete tests by the average access scale factor (1.7), and read tests by the bandwidth scale factor (1.8). The BSD-LFS create performance is approximately 25% worse than Sprite-LFS because BSD is not processor CPU bound, and the bandwidth scale factor is half the CPU scale factor. The BSD-LFS read performance dominates Sprite-LFS's due to larger tracks and track buffers. We cannot explain Sprite-LFS's delete performance.

Rosenblum's second benchmark evaluates the performance of large file I/O. This test consists of the following five passes through a 100 megabyte test file.

- Create the file by sequentially writing 8 KB units.
- 2. Read the file sequentially in 8 KB units.
- 3. Write 100 KB of data randomly in 8 KB units.
- 4. Read 100 KB of data randomly in 8 KB units.
- 5. Re-read the file sequentially in 8 KB units.

Figure 2 summarizes the results for the five file systems. All the sequential benchmarks and the LFS random write test are scaled by the bandwidth scale factor (1.8) and the remaining random tests are scaled by average access time scale factor (1.7). BSD-LFS and Sprite-LFS display nearly identical performance for the write tests. On the read tests, BSD-LFS displays superior performance due to its aggressive read-ahead and clustered I/O. During a sequential read, the file is read in 64 KB clusters and read-ahead is invoked on clusters rather than individual blocks. The read-ahead algorithm improves the BSD-LFS reread performance but degrades the random read performance. Although the file was written randomly, all the blocks within a segment are sorted. Since the eight kilobyte I/O requests consist of two logical blocks, read-ahead is always invoked on the second of

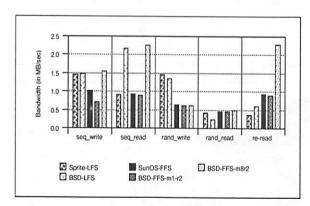


Figure 2. Validation of the BSD-LFS large file performance. The Sprite-LFS sequential read and write and random write results, and the Sun-FFS sequential read, write, and re-read results are scaled by the bandwidth scale factor (1.8). The random read and Sun-FFS random write results are scaled by the average access scale factor (1.7). The BSD-LFS implementation offers write performance equivalent to that of Sprite-LFS. BSD-LFS outperforms Sprite-LFS on sequential reads due to aggressive clustering and read-ahead. The read-ahead policy is also responsible for the degraded performance of random reads in BSD-LFS.

those blocks. Additionally, if logically sequential eight kilobyte units are written before the buffer cache is flushed to disk, the two eight kilobyte units (four blocks) will be allocated contiguously on disk. During the sequential re-read, BSD-LFS will read maximal-sized clusters based on disk layout.

In the random read case, the combination of readahead, the eight kilobyte I/O size, and the four kilobyte block size is a worst-case configuration for read-ahead. Two random eight kilobyte reads are actually four block reads to the file system. Assume that we are reading two random eight kilobyte units containing blocks 100, 101, 200, and 201. The read of block 100 is not sequential, so read-ahead is not triggered. However, when block 101 is read, the file system detects sequential access and triggers a read ahead operation on block 102. Unfortunately the next block read is block 200, requiring a seek. Since this is not a sequential access, read-ahead is not invoked on block 201. The end result is that we perform readahead when we do not want it (between eight kilobyte units) and we do not perform read-ahead when we do want it (after reading the first four kilobytes of an eight kilobyte unit). This phenomenon explains the low random read performance for BSD-LFS. The Sprite-LFS read numbers are consistent with an implementation that performs no read-ahead. Although the BSD-LFS random read performance is inferior to that of Sprite-LFS, none of the benchmarks analyzed in the remaining sections of this paper trigger the phenomenon described. The TPC-B benchmark discussed in Section 4 is the only benchmark that performs random I/O and its I/O size is equal to the file system block size, so the file system does not identify a pattern of sequential access.

Since Rosenblum did not set maxcontig high in Sun-FFS, we expect its performance to be comparable to BSD-FFS-m1r2 performance. This is consistent with all the results in Figure 2 except for the sequential write performance. To understand the performance for this test, we must deduce the rotdelay setting for the Rosenblum benchmarks. Typically, rotdelay is set to optimize write performance so that a disk revolution is not lost between successive Sun-FFS [6]. contiguous blocks approximately 40% of the possible disk bandwidth when writing sequentially. This is consistent with a rotdelay of one block. In BSD-FFS, our disks were sufficiently faster that we had to use a rotdelay of two blocks to avoid missing a rotation on every write. This yields write performance of slightly less than one-third the maximal bandwidth, which is consistent with our measurements.

The similarity in performance between Sprite-LFS and BSD-LFS demonstrates that BSD-LFS is an equivalently performing implementation of a log-structured file system. Its write performance is as good as Sprite's and it generally outperforms Sprite for reading due to the aggressive read-ahead and clustered I/O. Sun-FFS is a faithful implementation of the 4BSD FFS. The clustering modifications in 4.4BSD are a reimplementation of the modifications described by McVoy [7]. With these points established, we now compare the performance of BSD-LFS to that of the 4.4BSD FFS with these clustering enhancements. For the remainder of the paper, we use LFS to mean BSD-LFS and FFS to mean BSD-FFS.

3 Sequential Performance as a Function of File Size

Our first comparative benchmark examines the sequential read and write performance of the two file systems across a range of file sizes. The data set consists of 32 megabytes of data, decomposed into the appropriate number of files for the file size being measured. In the case of small files, where directory lookup time dominates all other processing overhead, the files are divided into subdirectories, each containing no more than 100 files. In the case of large files, we use either 32 MB or ten files, whichever generates more data. There are four phases to this benchmark:

- Create: All the files are created by issuing the
 minimal number of writes to create a file of the
 appropriate size. For all file sizes of four
 megabytes or less, this means that the test
 program does one I/O operation. For larger files,
 I/Os are issued in four-megabyte portions.
- Read: Each file is read in its creation order. The I/O operations are identical in size to those during the create test.
- Write: Each file is rewritten in its creation order.
- Delete: All the files are deleted.

The LFS measurements represent performance when there is no cleaner present and when the set of tests for each file size are run on a newly created file system. The FFS measurements represent empty file system performance; although a new file system is not created for each file size, all files from previous tests are deleted before the next data set is created. Both of these configurations yield the optimal operating conditions for the two file systems.

For all test results shown we provide the LFS performance and two FFS measurements. In both

cases, *maxcontig* has been set to 64 kilobytes (the maximum transfer size supported for our controller), but the *rotdelay* parameter varies. On our system, a *rotdelay* of two blocks provides the optimal write performance and a *rotdelay* of zero produces the optimal read performance. While a second write cannot be written before the first write completes, the track buffer caches read data so that the *rotdelay* is unnecessary. We show results for both *rotdelay* settings.

All measurements shown have confidence intervals of plus or minus one percent of the reported number.

3.1 Create Performance

Figure 3 shows the performance for the create phase of the sequential benchmarks. Since LFS buffers a large number of writes before writing any data to disk, the file creates in this benchmark occur asynchronously. In contrast, each time a *creat* system call returns, FFS guarantees that the file has been created and will exist after a system crash. The journaling file systems [4] avoid the synchronous writes of FFS by logging all meta-data operations to an auxiliary data structure, replacing multiple, random, synchronous I/Os with a single sequential one. When files are small the create test measures the ability of the systems to

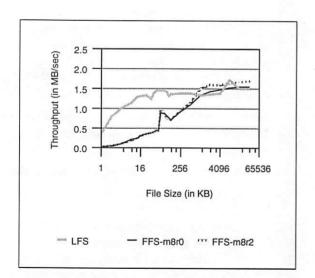


Figure 3. Create performance. When the speed of meta-data operations dominates (for small files less than a few blocks or 64 KB), LFS performance is anywhere from 4 to 10 times better than FFS. As the write bandwidth of the system becomes the limiting factor, the two systems perform comparably.

perform meta-data operations; when files are large, the create test measures the write performance of the file systems.

As expected, the asynchronous creation and the sequential writes of LFS yield superior performance to FFS for small files. The order of magnitude performance improvement advertised for LFS is demonstrated during the create benchmark when the file size is one kilobyte. In the two kilobyte to onehalf megabyte range, the superiority of LFS degrades from a factor of four at two kilobytes to a factor of two at 32 kilobytes to the same performance at onehalf megabyte. When the created files are large (onehalf megabyte or more) the performance of the two systems is comparable. The LFS and FFS-m8r0 systems use approximately 67% of the disk bandwidth, losing one rotation between each 64 kilobyte request. The FFS-m8r2 achieves approximately 75% of the disk bandwidth, losing only two blocks or two-thirds of one rotation between 64 kilobyte writes.

3.2 Read Performance

Figure 4 shows the read performance for both LFS and FFS as a function of the log of the file size. In the region between one and eight kilobytes both file systems show low bandwidth, but steady growth as the

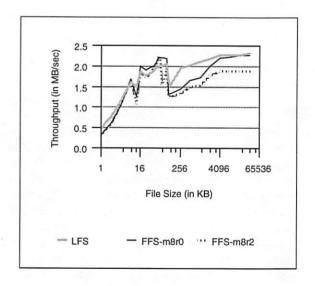


Figure 4. Read performance. For files of less than 64 KB, performance is comparable in all the file systems. At 64 KB, files are composed of multiple clusters and seek penalties rise. In the range between 64 KB and 2 MB, LFS performance dominates because FFS is seeking between cylinder groups to distribute data evenly.

transfer size between the disk and memory increases. Although the track buffer can hide disk latency, each file read results in a separate command to the device.

The dip in performance at sixteen kilobytes is due to the fact that two I/Os are required and read-ahead has not yet become operative. In the region between 8 KB and 64 KB, performance on both file systems improves as each approaches the 64 KB maximum I/O size. Files larger than 64 KB occupy more than one cluster and we see a performance dip when the next I/O operation is added.

FFS performance declines more steeply than LFS. There are two factors at work. FFS leaves a gap between clusters, the size of which is controlled by the *rotdelay* parameter. When *rotdelay* is non-zero and a new file is allocated, its blocks are placed in the gaps left by previously created files. Therefore, at file sizes greater than 64 KB, the files for FFS-m8r2 become increasingly fragmented. LFS does not suffer from this fragmentation since all of the blocks of a file are allocated contiguously. The ability to achieve the superior write performance of the FFS-m8r2 in the create test is precisely the limiting factor in the read case.

The second factor is the introduction of indirect blocks at files of 96 KB or larger. This drop is more pronounced for FFS since FFS begins allocation in a new cylinder group when the first indirect block is added. LFS continues allocating blocks sequentially.

There is a third parameter that can adversely affect FFS read performance in a test that creates many files. The FFS cpc (cylinders per cycle) parameter specifies the number of rotationally equivalent positions that will be considered in disk allocation. When a block is allocated, a preferred location is selected. If that location is unavailable, FFS will attempt to find up to cpc rotationally equivalent positions. This is exactly correct when allocating blocks to an existing file. However, when creating new files, the preferred block is set to the first block in a cylinder group. After the first file is created, that block is no longer available. Ideally, the next file should be allocated contiguously, but if cpc is nonzero, then the file will be allocated to a block that is rotationally equivalent to the first block in the cylinder group. Accessing these two blocks in succession requires a full rotation. Since the disk geometry of many of today's disks (e.g. SCSI) is not exposed, determining an accurate value for cpc is virtually impossible. The numbers reported here use a cpc of zero.

For large files, both file systems approach the 2.3 MB/sec bandwidth achievable over our SCSI bus.

3.3 Overwrite Performance

In this test, the 32 MB of data are written to the files already created in the create phase of the benchmark. The results are shown in Figure 5. In this test, FFS need perform no allocation, since the blocks are reused, but LFS must mark blocks dead and create new segments.

For files smaller than a cluster (the maximum I/O size supported by the controller, typically 64 KB), the sequential layout of LFS dominates. The performance drop for both systems at 64 KB is due to the cluster size. LFS and FFS-m8r0 lose a disk rotation between each pair of files, because although the files are allocated contiguously on disk, both systems must issue two separate I/O requests to write them. While track buffers can hide this effect during reading, they cannot diminish the effect while writing. The rotdelay gap alleviates this problem, but it introduces the performance penalty of fragmenting files whose blocks are allocated in these gaps. The 4.4BSD file system has been enhanced with a new reallocation algorithm that coalesces such fragmented files when they are created. However, the long term effects of this policy have not been thoroughly investigated, so we did not enable this functionality during the test.

For large files (greater than 256 KB), the onethird rotation that FFS-m8r2 saves over FFS-m8r0 and LFS accounts for the 15% performance improvement.

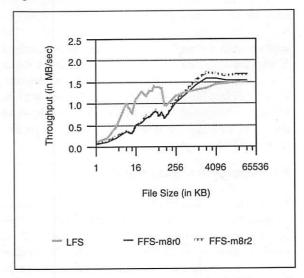


Figure 5. Overwrite performance. The main difference between the overwrite test and the create test is that FFS need not perform synchronous disk operations and LFS must invalidate dead blocks as they are overwritten. As a result, the performance of the two systems is closer with LFS dominating for files of up to 256 KB and FFS dominating for larger file sizes.

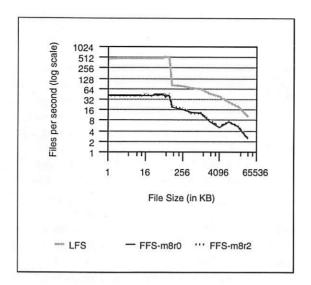


Figure 6. Delete performance. As was the case for creates, the delete performance is a measure of metadata update performance and the asynchronous operation of LFS gives it an order of magnitude performance advantage over FFS. As the file size increases, the synchronous writes become less significant and LFS provides a factor of 3-4 better performance.

3.4 Delete Performance

The final phase of this benchmark consists of the delete phase. Since this test writes little data, the results in Figure 6 are expressed in deleted files per second. Once again, the asynchronous behavior of LFS meta-data operations provides an order of magnitude performance improvement for small files. The sudden drop in performance occurs when indirect blocks are required. When a file exceeds its direct blocks, the indirect block must be retrieved from disk in order to free blocks (in FFS) or mark them dead (in LFS). Even for large file sizes, the asynchronous deletes of LFS provide three to four times the performance of FFS.

3.5 Benchmark Summary

To summarize these benchmarks, LFS offers an order of magnitude performance improvement in performing meta-data operations (creates and deletes) on small to medium-sized files. For deletes, LFS maintains its performance superiority at large file sizes, deleting at three to four times the rate of FFS.

The read, write, and create bandwidth for large files is comparable on both systems. FFS provides slightly better write performance when the rotdelay parameter is adjusted to avoid missing an entire rotation, and LFS provides slightly better performance reading since there is no effort to distribute the data across cylinder groups as is done in FFS.

For files smaller than the cluster size, read performance is comparable in both systems. LFS provides better write performance as its clustering of multiple small writes into large contiguous ones results in its using 43-65% of the available bandwidth.

As in all benchmarks, this one has many shortcomings. Both file systems are presented under optimal circumstances: all accesses are sequential, access order is identical to create order, the request stream is single-user, no cleaning is required for LFS, and FFS operates on an empty disk. The next section presents a more demanding workload, the TPC-B transaction processing benchmark.

4 Transaction Processing Performance

Although LFS was designed for a UNIX time-sharing workload, there has been speculation that the ability to convert small, random I/Os into large sequential ones would make it ideal for transaction processing [11]. Seltzer et al. measured a modified TPC-B implementation and found that the cleaning overhead severely limited its performance. The disk was 80% full in the benchmark configuration. In this section, we examine the performance of the same benchmark across a range of file system utilizations, since the file system utilization can affect cleaning cost. The benchmark configuration is identical to that described in Section 3, except that the file systems are configured with a four-kilobyte block size to match the block size of the database indexing structures.

The TPC-B benchmark simulates a check-cashing application [13]. The four files in the benchmark are described in Table 3. For each transaction, an account record is read randomly, its balance is updated, the balances in the associated teller and branch records are updated, and a history

File	Size	Description
account	237 MB	1,000,000 100-byte records
branch	44 KB	10 100-byte records
teller	44 KB	100 100-byte records
history	70 KB	Append-only; 50 bytes per transaction

Table 3: File specifications for a 10 TPS TPC-B database. Although our system is capable of supporting more than 10 TPS, we scaled the benchmark database to allow experimentation with disk utilization.

record is written. Although our implementation supports full concurrency control, we ran a single-user version of the test to minimize synchronization overhead and concentrate on disk behavior. All updates to the database are logged to a non-duplexed log residing on a separate disk.

The application maintains its own 4 MB block cache in user virtual memory. As the branch and teller files are rather small, they are always memory resident. In contrast, the account file is large, so only the internal pages of the B-tree index structure are likely to be memory resident. Therefore, the file system activity generated for each transaction is a random read of a block in the account file, followed by a random write from the user-level cache to the file system in order to make room for the newly retrieved account block. The newly retrieved account block is then dirtied and left in the cache until it is later evicted.

Three sets of results are shown in Figure 7. The top line indicates the performance of LFS in the absence of the cleaner. The performance was measured on the lowest utilization (48%) and projected across all utilizations since LFS will quickly run out of disk space if no cleaner is running. The second line on the graph shows FFS performance as a function of file system utilization. As expected, FFS shows no performance fluctuation as the disk becomes fuller. With the exception of the history file, every disk write in the benchmark merely overwrites an existing disk block, so there is no allocation and the fullness of the disk is irrelevant. Each data point represents the average of 100 iterations whose standard deviation was less than 1%.

In the absence of the cleaner, LFS provides approximately 50% better performance than FFS. The 50% performance difference can be attributed to LFS's ability to perform the random writes as sequential writes. In the LFS case, as dirty pages are evicted from the user-level buffer cache, they are copied into the file system cache. The dirty blocks remain in the cache until the number of dirty blocks exceeds a write threshold and LFS triggers a disk write. With the current system configuration, this triggering occurs when 115 blocks have accumulated (representing 115 transactions). These transactions progress at a rate limited by the time required to randomly read the account records from disk. To read, we must copy the page being read from the kernel into the user cache and must also evict a page from the user cache, copying it into the kernel. On our system, these two copies take approximately 1.8 ms. With a 9.5 ms average seek, a 5.5 ms average rotational delay, and a 1.6 ms transfer time, each random read

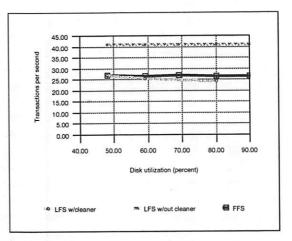


Figure 7. Transaction processing performance. While LFS can potentially provide dramatically improved performance, when the cleaner runs, its performance is comparable to FFS. The performance is largely independent of the disk utilization as the steady-state cleaning overhead is dominated by segment read time.

requires 18.4 ms for a throughput of 54 transactions per second. Next, the segment must be flushed. The 115 data blocks are likely to have caused the 58 indirect blocks in the account file to be dirtied, so our segment contains 115 data blocks, 58 indirect blocks, one inode block, and one segment summary for a total of approximately 700 KB. Using the bandwidth numbers from Section 3.3, we can write the 700 KB at a rate of 1.3 MB/sec for a total time of 0.5 seconds. Therefore, processing 115 transactions requires $115 \times 18.4 + 500$ ms yielding 44 transactions per second, within 7% of our measurement.

The calculation for FFS is much simpler: throughput is limited by the performance of the random reads and writes. Each random I/O requires a 9.5 ms seek, a 5.5 ms rotation, a 0.9 ms copy, and a 1.6 ms transfer for a total of 17.5 ms yielding throughput of 28.6 transactions per second, within 7% of our measurement.

The introduction of the cleaner changes the results substantially. At a file system utilization of 48%, LFS performance is comparable to FFS, yielding a performance degradation of 34% due to the cleaner. As the disk becomes more full, the impact increases to the approximately 41% degradation, as observed in [10]. The good news is that this performance is comparable to FFS, but the unfortunate result is that any performance advantage of LFS is already lost at a file system utilization of only 48%.

To understand the LFS performance, we must examine the operation of the cleaner and its interaction with the benchmark. In the steady state, each segment dirtied by LFS requires that the cleaner produce one clean segment. If we were insensitive to response time and wished only to clean most efficiently, we would run the benchmark until the disk filled, then clean, and then restart the benchmark. This should produce the best possible throughput in the presence of the cleaner.

As discussed earlier, LFS fills segments at a rate of 115 transactions per 700 KB or 168 transactions per segment. For simplicity, call the database 256 MB and the disk system 512 MB. This requires 256 segments, 43,000 transactions, or 1000 seconds at the "no-cleaning" LFS rate. After the 1000 seconds have elapsed, LFS must clean. If we wish to clean the entire disk, we must read all 512 segments and write 256 new ones. Let us assume, optimistically, that we can read segments at the full bus bandwidth (2.3 MB/sec) and write them at two-thirds of the disk bandwidth (1.7 MB/sec), missing a rotation between every 64 KB transfer. The cleaning process will take 223 seconds to read and 151 seconds to write for a total of 374 seconds. Therefore, at 50% utilization our best case throughput in the presence of the cleaner is 31.3 transactions per second. This is within 15% of our measured performance.

Unfortunately, LFS cannot clean at the optimal rate described above. First, the transaction response would be unacceptably slow while the cleaner stopped for six minutes to clean. Secondly, the calculations above assumed that the disk is read sequentially. Since the selection of segments is based on Rosenblum's cost-benefit algorithm [9], there is no guarantee that collections of segments being cleaned will be contiguous on disk. Thirdly, the history file in the benchmark grows by 50 bytes per transaction, so file system utilization increases during the test run. Finally, cleaning requires that multiple segments be cached in memory for processing, so we must limit the number of segments cleaned simultaneously.

Since LFS cannot clean at its maximal rate, it should clean at a rate that permits it to perform its segment reads and writes at near-optimal speed. At 50% utilization, we should be able to read two dirty segments and produce one clean segment. Reading one megabyte requires a random seek (9.5 ms) one-half rotation (5.5 ms) and a 1 MB transfer (435 ms) for a total of 450 ms per segment read. Rewriting the segment requires the same seek and rotation, but the transfer requires 588 ms for a total of 603 ms for the write or 1.5 seconds to clean the segment. In the steady state, this cleaning must be done for each 168

transactions. Our throughput without the cleaner is 41 transactions per second, so it takes 4.1 seconds to execute 168 transactions and 1.5 seconds to clean, yielding 5.6 seconds or 30.0 TPS. This is within 10% of our measured number.

It can be argued that LFS loses performance because it writes indirect blocks too frequently (approximately once every three seconds in our benchmarking environment). The current BSD-LFS write policy assumes that when the number of dirty buffers in the cache exceeds the write threshold (the point at which the kernel triggers a segment write), generating clean buffers is essential and it uses an aggressive policy of writing all the dirty buffers to disk. If the dirty indirect blocks were cached during this benchmark, the number of dirty data blocks that are allowed to accumulate in the cache would be reduced and segment writes would occur more frequently. While suboptimal for this benchmark, we believe that flushing indirect blocks with their data blocks is the correct default policy.

5 Effects of Free Space Fragmentation on FFS Performance

Both LFS and FFS rely on the allocation of contiguous disk blocks to achieve high levels of performance. Because the results in Section 3 were obtained from newly-created, empty file systems, there was no shortage of contiguous extents of free space. On real systems, in use for extended periods of time (months or years), the file system cannot expect to find such an ideal arrangement of free space. LFS and FFS deal with this reality in two different ways, both of which cause performance overhead for the respective file systems.

LFS relies on the cleaner process to garbage collect old segments, creating large regions of free space—clean segments. The cleaner imposes overhead on the overall performance of LFS. Section 4 discusses this overhead in the context of a transaction processing workload.

In contrast, FFS makes no assumptions about the layout of free space on the file system. FFS uses whatever free space is available on the disk, contiguous or not. In fact, the block allocation policy of FFS remained unchanged when clustering was added [7]. FFS may not allocate contiguous blocks to a file, even when contiguous free space is available. As with the LFS cleaner, this may adversely effect performance.

The fragmentation of free space in an FFS may increase with time or with file system utilization. This fragmentation can degrade performance as an FFS

ages. To assess this risk, we studied a collection of FFS file systems on the file servers of the Division of Applied Science at Harvard University over a period of nine months to examine how the performance of FFS file systems under real workloads differs from the performance of the empty FFS file systems typically used for benchmarking.

5.1 Data Collection

We collected data from fifty-one file systems on four file servers. All of the file servers were SparcStations running SunOS 4.1.3. Although the operating system is substantially different than the 4.4BSD operating system used elsewhere in this study, the file systems are nearly identical. (The BSD-FFS clustering enhancements were based on those originally implemented in SunOS [7].)

Our data collection consisted of daily *snapshots* recorded for every file system under study. A snapshot is a summary of a file system's meta-data, including information about the size and configuration of the file system, the age, size, and location of each file, and a map of the locations of free blocks on the file system.

In the interest of brevity, the presentation here is limited to six representative file systems. The remaining file systems in the study demonstrated behavior similar to one or more of these. The important attributes of these file systems are summarized in Table 4.

5.2 Data Analysis

A separate study performed extensive analysis of this snapshot data [12]. Examining this data in conjunction with the FFS block allocation algorithm provided a variety of interesting information characterizing the layout of FFS file systems. Some of the important results are summarized here.

An evaluation of the FFS block allocation algorithm showed that when a new file is created, FFS attempts to allocate space for it starting from the beginning of its cylinder group. The first free block in a cylinder group is almost always allocated as the first block of a new file. FFS attempts to extend the file according to the constraints of the *maxcontig* and *rotdelay* parameters. In practice, this means that as a file grows, it uses free blocks in order from the beginning of the cylinder group. File systems with a non-zero *rotdelay* may occasionally skip blocks as dictated by this parameter. All of the file systems studied here had a *rotdelay* of zero.

Because of this allocation pattern, free space within a cylinder group tends to be unevenly distributed, with most of the free space located at the end of the cylinder group. Furthermore, the free space near the beginning of a cylinder group is more fragmented than the free space near its end. This uneven distribution of free space, combined with the FFS block allocation algorithm, causes small multiblock files to be more fragmented than larger files. When a small file is created, FFS allocates space to it starting from the beginning of a cylinder group. This space is likely to be highly fragmented, resulting in a fragmented file. Space for larger files is also allocated from the beginning of a cylinder group, but a large file

Server	Size (MB)	Age (months)	ncg	bpg	Descriptions
das-news	565	31	49	1539	News articles and software
virtual12	1705	24	89	2596	Installed software and sources
das-news	353	30	31	1539	System Administrators' accounts
speed	953	17	85	1520	User accounts
endor	953	14	85	1520	Course accounts; 1st-year grad accounts
endor	450	14	40	1520	User accounts for theory
	das-news virtual12 das-news speed endor	Server (MB) das-news 565 virtual12 1705 das-news 353 speed 953 endor 953	Server (MB) (months) das-news 565 31 virtual12 1705 24 das-news 353 30 speed 953 17 endor 953 14	das-news 565 31 49 virtual12 1705 24 89 das-news 353 30 31 speed 953 17 85 endor 953 14 85	Server (MB) (months) ncg bpg das-news 565 31 49 1539 virtual12 1705 24 89 2596 das-news 353 30 31 1539 speed 953 17 85 1520 endor 953 14 85 1520

Table 4: File system summary for FFS fragmentation study. All file systems had an eight kilobyte block size, *maxcontig* of seven and *rotdelay* of 0. The Age column indicates the file system age as of November 1, 1994. Subtract nine months to obtain the age at the beginning of the study. The ncg and bpg columns indicate the number of cylinder groups and the number of file system blocks per cylinder group, respectively.

is more likely to be able to take advantage of the well clustered free space later in the cylinder group.

The data collected by the snapshots showed that only 37% of the two block files allocated on twelve different file systems were allocated to two consecutive blocks on the file system. In contrast, 80% of the blocks allocated to 256K (32 block) files were contiguous.

The amount of fragmentation in a file is of critical importance because it is one of the primary factors determining file system performance when reading or writing the file.

5.3 Performance Tests

Because the SunOS file systems we studied were in active use, it was not feasible to run benchmarks on them. Instead, the meta-data in the snapshots was used to reconstruct the file systems on the disk of a test machine. This not only allowed the file systems to be analyzed in an otherwise quiescent environment, but also made it easier to study the performance of comparable empty file systems. In the following discussion, the term "original file system" is used to refer to the actual file systems on the file servers, and "test file system" or "copied file system" is used for the file systems reproduced on the test machine. The benchmarks described in this section were run on a SparcStation I with 32 megabytes of memory, running 4.4BSD-Lite. The details of the system are summarized in Table 5.

In order to approximate the original file systems' configurations as closely as possible, the test file systems were created using the same values of *rotdelay* and *maxcontig*. The test file systems were also configured with the same number of cylinder groups as the corresponding original file systems. When different sized cylinders on the original and test

CPU Parameters			
CPU	SparcStation I		
Mhz	20		
Disk Parameters			
Disk Type	Fujitsu M2694EXA		
RPM	5400		
Sector Size	512 bytes		
Sectors per Track	68-111		
Cylinders	1818		
Tracks per Cylinder	15		
Track Buffer	512 KB		
Average Seek	9.5 ms		

Table 5: Fragmentation benchmark configuration.

disks made it impossible to precisely duplicate the size of the cylinder groups, slightly larger cylinder groups were created on the test file system. The extra blocks in each cylinder group were marked as allocated, so the test file system could not use them.

Because all of our file system benchmarks rely on the creation and use of new files, the most important characteristic of the original file systems to reproduce is the arrangement of free space. Other details, such as the precise mapping of allocated blocks to files, and the directory hierarchy, are less important, because they have little or no impact on the layout or the performance of newly created files. Therefore the only meta-data copied from the snapshots of the original file systems was the free space bitmap for each cylinder group. The resulting test file system contained the root directory and no other data, but could only utilize the same free blocks as the original file system.

The benchmarks from Section 3 were used to compare these reconstructed file systems to comparable empty file systems. The latter were created in the same manner described above, but their free space bitmaps were not modified. The create and read phases of the sequential I/O benchmark were used to analyze the performance of the file systems. One minor modification was made to the benchmark program for these tests. Instead of creating one hundred files per directory, the benchmark used here creates only twenty-five files per directory. This configuration stresses the file system more and the CPU less since it reduces the processing overhead in finding directory entries and causes FFS to create more directories, which are distributed among the cylinder groups.

These results presented here show the performance for three different file sizes (eight kilobytes, sixty-four kilobytes, and one megabyte) and four different dates spread over the nine month measurement period.

5.4 Test Results

For each file size tested, the throughput for reading and creating the copied file systems was compared to the throughput for the corresponding empty file system. The graphs in Figure 8 show the performance of each of the test file systems as a percentage of the throughput of the corresponding empty file system. The disk utilization (percentage of total blocks that are allocated) for each file system is also displayed.

Ultimately, the performance of these benchmarks depends on the layout of the test files. Files that are laid out contiguously on the disk can be read and

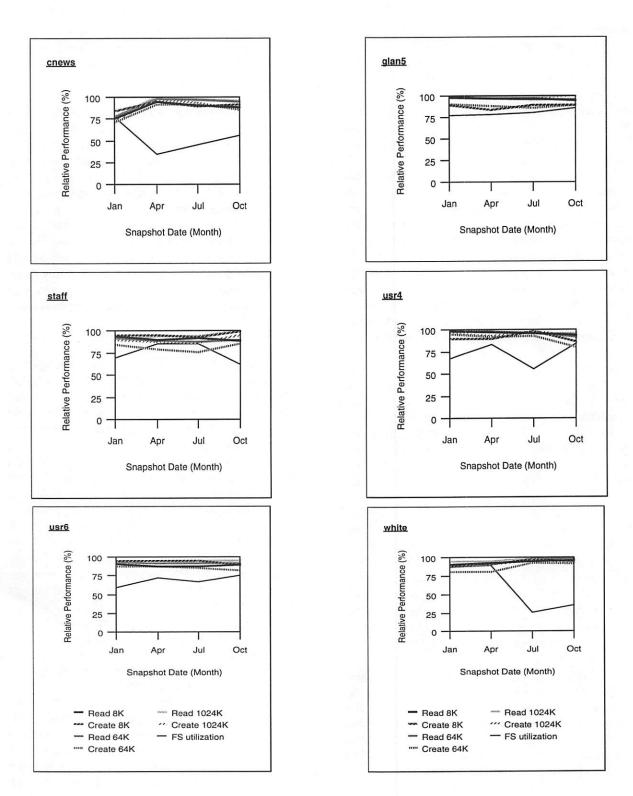


Figure 8. Effect of fragmentation on FFS performance. The graphs display the performance of the six test file systems as a percentage of the performance of a comparable empty FFS on the same disk. The utilization of each file system for each test date is also shown. The benchmarks were run using snapshots of the file systems taken on the 26th day of the designated months (except for the October snapshots for *cnews* and *staff*, which were taken on the 20th). Most file systems exhibit little deterioration over the time period, performing at 85-90% of the empty file system performance. Where changes occur, they often correlate to utilization (e.g. *cnews*, *staff*, *white*, *usr4*). We do not yet see a correlation between file system age and performance degradation.

written quickly. In contrast, files whose blocks are scattered across the disk do not perform as well. Achieving optimal file layout requires contiguous free space on the file system. Thus, anything that causes free space to become fragmented will degrade the performance of a file system. There are three factors that may contribute to the fragmentation of free space in an FFS: high file turnover, high utilization, and file system age.

The news partition, *cnews*, is an example of a file system with high turnover. As a repository for netnews articles, most of the files on this file system are extremely small, and there is a high rate of file turnover. This should cause its free space to rapidly become fragmented. This hypothesis is supported by the fact that approximately two weeks after the January copy of this file system was made, *cnews* reported that it was out of disk space, but an examination of the file system showed that although there were no free blocks, there was 92 megabytes of free space (16% of the file system)—all of it in fragments.

Not surprisingly, the January tests on *cnews* exhibited the greatest performance differences of any of the file systems. For five out of the six test cases, performance was less than 80% of that of the empty *cnews* file system. The single greatest performance difference in all of the tests was in the 64 KB write test on cnews. This test achieved only 71% of the empty file system performance.

In contrast, *glan5* is a file system with very little file turnover. It is used to store the sources and binaries for various software packages. Thus, files on this file system are created or deleted only when new software is installed or when existing software is upgraded. Not surprisingly, *glan5* performed better than the other file systems. Fifteen of the twenty-four tests on copies of *glan5* achieved 95-100% of the performance of the empty *glan5* file system.

that The second factor contributes fragmentation is high disk utilization. A highly utilized file system has few free blocks, which are unlikely to be well-clustered. The effect of disk utilization on file system performance is demonstrated by several of the file systems in Figure 8. Nearly all of the large changes in utilization are accompanied by inverse changes in performance (see cnews and staff for particularly noticeable correlation). Although performance drops as the file systems become full, the performance can be regained by removing a fraction of the files.

The third parameter affecting FFS fragmentation is the age of the file system. Even if a file system has a light workload with little file turnover, after several

years the cumulative effect of these small changes should be comparable to a high file turnover on a younger file system. The data provides some evidence for this phenomenon, but it is inconclusive. The two oldest file systems (cnews and staff) are the only file systems where performance on the copied file systems was less than 80% of the performance of the empty file system. As discussed above, however, part of the cnews performance should be attributed to its usage pattern rather than its age. Although several test cases performed poorly on the staff file system, other tests on same date performed as well on staff as on the younger file systems. In our department, file systems "turnover" approximately every three years. That is, disks are replaced, so the data are dumped and restored. An informal poll indicated that this threeyear turnaround is fairly typical, and coincides with IRS regulations concerning equipment depreciation.

5.5 File Size and Performance

The graphs in Figure 8 indicate that there are performance differences for the different file sizes tested. Many of the largest differences between the copied file systems and empty file systems occur in the 64 KB test. The 8 KB and 1024 KB were less sensitive to the file system age, utilization, and turnover rate.

For each of the six file systems, we ran read and create tests for each of three file sizes on four different dates for a total of 144 tests. Of the forty-eight tests that used a 1024 KB file size, only two performed at less than 85% of the corresponding empty file system's bandwidth. These two points were for January on the *cnews* partition. Excluding the January tests on *cnews*, which were generally worse than any other test, of the forty-six tests that used a file size of 8 KB, there was only one test case that achieved less than 85% performance (the April write test on *glan5*, 84.2%). Of the forty-eight tests using 64 KB files, nine performed at less than 85% of the empty file system's throughput. Of the remaining thirty-nine, thirty-five of them were over 90%.

There are a variety of reasons for the performance differences among the different file sizes. All of the file sizes suffer from the increased fragmentation of free space on the copied file systems. This is most noticeable in the 64 KB files because of ameliorating circumstances in the 8 KB and 1024 KB tests.

The large size of the 1024 KB files allows FFS to perform more read-ahead than for the smaller file sizes. Read-ahead helps to offset the cost of fragmentation by initiating a read (and possibly an accompanying seek) before the data is needed. FFS

benefits from read-ahead within a 64 KB file, but does not perform predictive reading of different files. However, on the empty file system, the beginning of the next file will be in the disk's track buffer (if the file is in the same cylinder group and begins on the same track) and can be accessed rapidly. On the copied file system, a seek is often required to access the first block of the next file. This seek is not initiated until the benchmark program has issued the corresponding read system call. The 1024 KB file benchmark does not demonstrate this phenomenon because FFS changes to a new cylinder group after allocating the first twelve blocks of a file. In both the empty and copied file systems, a seek is required to move from the end of one file to the beginning of the next.

The fragmentation of free space has little impact in the 8 KB test case. The benchmark creates directories containing twenty-five files. Each directory is placed in a different cylinder group. Since the block size is 8 KB for all of the file systems, the 8 KB test case will read or write twenty-five blocks of data to one cylinder group, then seek to a different cylinder group to start the next directory. Thus, the 8 KB test spends a larger portion of its time performing seeks than either of the other test sizes. Increased fragmentation in the copied file systems means that the files in a given directory are typically spread out more than on an empty file system. Because of the large number of seeks between cylinder groups, however, the amount of overhead introduced by this fragmentation has a smaller impact on overall performance than in the 64 KB test case.

5.6 Benchmark Summary

Performance tests using real-world file systems indicate that there is justification for concerns over the performance impact of fragmentation in FFS. Such concerns should be mitigated, however, by many of the results of these tests. Although the greatest performance differences between a real file system and an empty one were almost 30%, most file systems showed far smaller differences, especially for large files. According to Baker et al. [1], over half the bytes transferred to/from disk come from files over 1 MB in size. The measured performance differences for such files is less than 15% and in 70% of our tests, large files performed at 95% or better on the copied file systems.

It is worth noting that many of the greatest performance differences between real and empty file systems occurred on a file system that demonstrates worst case behavior in almost every way. The *cnews* file system holds small, rapidly replaced files. This file system suffered an unusual fragmentation failure right after these large performance differences were noted. It is difficult to imagine a file system that would incur a greater fragmentation penalty.

6 Conclusions and Future Work

Our results show that the comparison of FFS and LFS is not an easy one. FFS performance can be modified substantially by tweaking parameters such as *maxcontig*, *rotdelay*, and *cpc*.

Unquestionably, when meta-data operations are the bottleneck, LFS provides superior performance to FFS. When creating files of one kilobyte or less, or when deleting files of 64 KB or less, LFS provides an order-of-magnitude performance improvement. This improvement comes in part from LFS's disk layout, and in part from the asynchronous implementation of these operations. There are several alternatives for providing asynchronous meta-data operations, including journaling file systems [4] and ordering updates. (Using an ordered update approach, Ganger reports a factor of five to six improvement in FFS meta-data update performance [3].)

When LFS cleaner overhead is ignored, and FFS runs on a new, unfragmented file system, each file system has regions of performance dominance.

- LFS is an order of magnitude faster on small file creates and deletes.
- The systems are comparable on creates of large files (one-half megabyte or more).
- The systems are comparable on reads of files less than 64 kilobytes.
- LFS read performance is superior between 64 kilobytes and four megabytes, after which FFS is comparable.
- LFS write performance is superior for files of 256 kilobytes or less.
- FFS write performance is superior for files larger than 256 kilobytes.

Cleaning overhead can degrade LFS performance by more than 34% in a transaction processing environment. Fragmentation can degrade FFS performance, over a two to three year period, by at most 15% in most environments but by as much as 30% in file systems such as a news partition.

There is more work to be done. The effects of cleaning on LFS in other environments are still not

fully understood. Trace analysis indicates that in a network of workstations environment, there may be sufficient idle time that cleaning can be accomplished without I/O penalty [2].

7 Availability

The file system and benchmark source code and trace data are available and freely redistributable. Send electronic mail to margo@das.harvard.edu.

8 Acknowledgments

Many people helped make this paper possible. We would like to thank Carl Staelin and Kirk McKusick who provided considerable help in maintaining our systems at Berkeley; Janusz Juda who ran all our data gathering scripts for months on end; John Ousterhout who originally suggested this work and offered much advice and criticism; Ken Lutz who produced disk specifications with amazing rapidity, and Diane Tang, Larry McVoy, John Wilkes, and David Patterson who made many useful suggestions about our presentation.

9 References

- [1] Baker, M., Hartman, J., Kupfer, M., Shirriff, K., Ousterhout, J., "Measurements of a Distributed File System," *Proceedings of the 13th Symposium on Operating System Principles*, Pacific Grove, CA, October 1991, 192-212.
- [2] Blackwell, T., Harris, J., Seltzer, M., "Heuristic Cleaning Algorithms in LFS," Proceedings of the 1995 Usenix Technical Conference, New Orleans, LA, January 1995.
- [3] Ganger, G., Patt, Y., "Metadata Update Performance in File Systems," Proceedings of the First Usenix Symposium on Operating System Design and Implementation, Monterey, CA, November, 1994, 49-60.
- [4] Howard, J., Kazar, Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, N., West, M., "Scale and Performance in a Distributed File System," *ACM Transaction on Computer Systems* 6, 1 (February 1988), 51-81.
- [5] Lieberman, H., Hewitt, C., "A real-time garbage collector based on the lifetimes of objects," *Communications of the ACM*, 26, 6, 1983, 419-429.
- [6] McKusick, M.Joy, W., Leffler, S., Fabry, R. "A Fast File System for UNIX, "ACM Transactions on Computer Systems 2, 3 (August 1984), 181-197.

- [7] McVoy, L., Kleiman, S., "Extent-like Performance from a UNIX File System," *Proceedings of the 1990 Summer Usenix*, Anaheim, CA, June 1990, 137-144.
- [8] Rosenblum, M., Ousterhout, J., "The LFS Storage Manager," *Proceedings of the 1990 Summer Usenix*, Anaheim, CA, June 1990, 315-324.
- [9] Rosenblum, M., Ousterhout, J., "The Design and Implementation of a Log-Structured File System," ACM *Transactions on Computer Systems* 10, 1 (February 1992), 26-52.
- [10] Seltzer, M., Bostic, K., McKusick., M., Staelin, C., "The Design and Implementation of the 4.4BSD Log-Structured File System," *Proceedings of the 1993 Winter Usenix*, January 1993, San Diego, CA.
- [11] Seltzer, M., "Transaction Support in a Log-Structured File System," Proceedings of the Ninth International Conference on Data Engineering, Vienna, Austria, April 1993.
- [12] Smith, K. A., Seltzer., M, "File Layout and File System Performance," Harvard Division of Applied Sciences Technical Report, 1994.
- [13] Transaction Processing Performance Council, "TPC Benchmark B Standard Specification," Waterside Associates, Fremont, CA., August 1990.

Metadata Logging in an NFS Server

Uresh Vahalia, EMC Corporation¹
Cary G. Gray, Abilene Christian University
Dennis Ting, EMC Corporation

ABSTRACT

Over the last few years, there have been several efforts to use logging to improve performance, reliability, and recovery times of file systems. The two major techniques are metadata logging, where the log records metadata changes and is a supplement to the on-disk file system, and logstructured file systems, whose log is their only ondisk representation. When the file system is mainly or wholly accessed through the Network File System (NFS) protocol, it adds new considerations to the suitability of the logging technique. NFS requires that all operations be updated to stable storage before returning. As a result, file system implementations that were effective for local access may perform poorly on an NFS server. This paper analyzes the issues regarding the use of logging on an NFS server, and describes an implementation of a BSD Fast File System (FFS) with metadata logging that performs effectively for a dedicated NFS server.

1. Introduction

Recent years have seen improvements in CPU speeds that have not been matched by comparable improvements in disk access speeds. As a result, disk I/O has become the new bottleneck in operating system performance [Oust 90]. Traditional file systems perform poorly when run on fast machines with relatively slow disks. This has motivated the development of new file systems that seek to reduce the frequency and latency of disk access. One important technique is to use logging, which holds the promise of higher performance, greater reliability, and quick crash recovery.

Transparent access to files on other machines is a relatively new development. Consequently, most file systems are designed primarily for local access. When the same file system is used for exporting files and directories over a network, many assumptions inherent in its design are invalid, and the file system may perform poorly. In particular, the *Network File System* (NFS) protocol [Sand 85] requires the server to commit any file system modification to stable storage before returning the results of a request. Consequently, an NFS file system must perform many more disk writes, most of them synchronous, than a file system used for local access.

Recent implementations of logging in file systems require extensive changes both to the kernel algorithms and to the on-disk structures. Development costs are high, since not only the kernel code, but also utilities such as *newfs(8)* and *dump(8)* must be rewritten. To upgrade to the new file system, the system administrator must back up all partitions, boot the new kernel, reformat the disks, and finally restore all files. Their performance gains, even for local access, are not significant compared with other optimizations such as file system clustering [McVo 91]. Moreover, current logging file systems are optimized for local access, and their advantages are reduced for NFS use.

The Calaveras project [Rama 94] was an advanced development effort at Digital Equipment Corporation. Its aim was to build a dedicated, high-performance, multi-protocol file server using commodity hardware and conforming to existing software standards. Its first prototype was an NFS server running on Intel platforms. While designing the Calaveras file system, we wanted to gain the advantages of logging, namely high reliability and quick crash recovery, while retaining the on-disk layout of the BSD Fast File System (FFS) [McKu 84]. It was also critical to optimize the file

 $^{{\}bf 1}$ The work in this paper was performed at Digital Equipment Corporation.

system for NFS-only access, which required addressing some of the drawbacks of existing logging implementations.

We describe here the implementation and performance of the Calaveras file system, which uses metadata logging to enhance a traditional FFS.

2. Background

The original UNIX file system [Thom 78] uses simple disk layout and algorithms, but performs very poorly. It uses small, fixed-size blocks that are allocated randomly from the disk. FFS, introduced in the 4.2BSD release, provides a major improvement. FFS uses large block sizes (typically 4K or 8K bytes), and tries to optimize disk access by intelligent placement of blocks. It divides the disk into cylinder groups, comprising a set of contiguous cylinders. Each cylinder group stores file data as well as metadata (inodes, directories, and indirect blocks), and the allocation algorithm tries to place related information in the same cylinder group. It also tries to minimize rotational delays by predicting the amount of disk rotation between consecutive read operations (the rotational delay, or rotdelay), and separating successive blocks of the same file by that amount.

There are several limitations performance. The rotdelay factor optimizes for the case when the file is accessed one block at a time. At the same time, it limits the disk performance to a fraction of raw disk access speed. For instance, if the rotdelay for a disk is set to one (the best case), two successive blocks of a file are separated by one unrelated block. This limits the disk bandwidth to half its raw capacity. Secondly, many file operations require several different I/O operations, some of which have to be done synchronously to preserve file system consistency. For example, a file create operation allocates and writes a new inode, and modifies the parent directory and its inode. This requires three disk accesses (more if the blocks are not already in memory), each of them potentially involving a head seek.

Further improvements have come in three major directions. One is to modify file system algorithms to reduce and optimize disk accesses, such as I/O clustering [McVo 91] and write-gathering of NFS operations [Jusz 94]. The second is to use non-volatile read-only memory (NV-RAM) [Mora 90, Hitz 94] to delay and batch writes that normally must be synchronous. The third is to use logging as either a supplement or a substitute for normal file system writes.

The logging technique is particularly attractive since its benefits are not restricted to performance. It offers increased reliability, since the log may replicate some or all of the file system data and metadata. It also allows quick recovery after a system crash, since a log playback is usually much faster than the file system checking and patching performed by fsck(8) [Kowa 78] in traditional systems. This is an extremely important consideration for installations that require high availability and cannot tolerate long delays due to crashes.

2.1. Logging file systems

A file system can use logging in two ways - it can be log-structured, or log-enhanced. The former approach represents the entire file system as a single, continuous log [Finl 87, Oust 89]. It relies on a large cache to handle most read requests, and tries to write the log in large chunks, sequentially on the disk. Operations are batched as far as possible to avoid small writes. Whenever a block is modified, it is simply rewritten to the tail of the log instead of updating in place, and other data structures are updated and similarly rewritten to reflect the new position. In time, many blocks in the log become invalid, either because they have been rewritten further ahead in the log, or because the corresponding files have been deleted. system tracks and garbage collects these blocks, freeing up space needed when the disk is full and the log must wrap around. This requires compacting the free space, rewriting scattered active blocks to the head of the log.

[Selt 93] describes an implementation of a logstructured file system (LFS) for 4.4BSD UNIX, based on similar work for Sprite in [Rose 91]. While LFS performs better than traditional FFS, its performance gains are matched, and in some ways bettered, by simpler enhancements to FFS such as the file system clustering work of [McVo 91]. LFS also involves a major code rewrite and on-disk structures that are incompatible with FFS. Moreover, the performance benefits of LFS come directly from the ability to write the log in large chunks. This is generally not possible with NFS, which requires synchronous commits. Finally, the garbage collection and compaction substantially reduce the performance; in some benchmarks, the performance of LFS with garbage collection was as much as 20% worse than standard FFS.

In log-enhanced file systems, the log is a supplement to, and replicates information in, the normal on-disk structures [Hagm 87]. Typically, the log only records changes to metadata objects (inodes, directories, allocation maps, etc.), perhaps in an ordinary file in the same file system. If the system is shutdown gracefully, the log can be discarded, since the file system is up to date and consistent. In the event of a crash, however, the log is used to rebuild the file system. During normal operation, each metadata write is first written synchronously to the log, and the on-disk structures are updated later during cache flushes. Hence after a crash, the ondisk structures may contain stale data, but the log has a record of all completed operations, and can be played back to recover the file system to a consistent state.

Metadata logging may improve performance as well. On one hand, each metadata update is written to disk twice - once to the log, and once to its normal location on disk (we call this write the inplace update). On the other hand, since the in-place updates are delayed, they are often eliminated or batched. For example, the same inode may be modified several times before it is flushed, and multiple inodes in the same disk block are written out together. The log writes are batched as well. For a single operation such as create, the changes to the directory and the two inodes can be combined in a single log entry. Multiple operations that are temporally close to each other can be similarly batched. This reduces the total number of disk writes for metadata blocks. The overall impact on performance depends on the ratio of the metadata operations (such as create, delete, and link) to data writes. If much of the activity in a system is large file writes, the performance improvement is negligible.

2.2. Considerations for NFS access

The behavior and performance of a file system are very different when it is accessed locally and when it is accessed by remote clients using a file access protocol such as NFS. NFS is a stateless protocol, and neither the server nor the clients are required to maintain state information about the other (although both usually maintain some state for performance reasons). When a server crashes and recovers, the client has no way of knowing it, and the effect to the client is similar to that of a network delay. For such a protocol to work consistently, the server is required to commit all file system modifications to stable storage before returning the results of an operation.

This condition has a great impact on file system behavior. For local file systems, the kernel delays most disk writes until it needs to flush its cache. This has many advantages. Multiple writes to the same block between cache flushes are all committed by a single disk write. The disk driver can effectively reorder the writes to minimize head seeks. Many writes can be eliminated altogether – if a user creates a temporary file, writes some data into it, and deletes it shortly thereafter, the data blocks may never be written to disk.

NFS requests, on the other hand, require frequent synchronous disk writes. Each write request that increases the size of a file causes at least two disk writes — one to write the data block, and one to write the updated inode. Additional writes are necessary if an indirect block must be created or modified. NFS access thus generates a lot more disk I/O and offers less room for traditional optimizations such as reordering of disk requests. Consequently, a file system that is suitable for local access may perform poorly if used mostly or wholly for NFS access.

NFS server performance is characterized by two metrics – latency and throughput. Latency measures the average time taken for each NFS request, while throughput is the maximum load the server can bear, measured in NFS operations per second. The two are interrelated; as the load on a server increases, so does the average latency. There is also room in the design for a tradeoff. A log-structured file system, for instance, achieves high performance by batching writes. This results in increased throughput. On the other hand, since the write must be committed to disk before replying to the NFS request, such batching can result in unacceptably high latency.

High latency causes several problems. Users see the system as slow and unresponsive. If a request takes very long to complete, the client may time out and retransmit the request. If the server cannot detect or handle these retransmissions effectively, it may perform a lot of duplicate processing. This further escalates the performance degradation, and also causes numerous correctness and consistency problems [Jusz 89].

3. Calaveras file system design

We began by porting the UNIX file system (ufs)² from DEC OSF/1 to the Calaveras kernel. This required some changes since the storage interface,

 $^{^2\}text{ufs}$ refers to the implementation of FFS under the vnode/vfs interface [Klei 86]

scheduling, and buffer cache of Calaveras [Rama 94] were different from OSF/1. The on-disk structures were identical to those of FFS. Once we had a working prototype, we decided to enhance it by adding metadata logging.

The primary motivation for logging was to provide quick crash recovery. We wanted to eliminate the need for *fsck*, which can take tens of minutes on a file server with a large number of disks. We also wanted equal or better performance – in particular, it was essential that the addition of logging not increase the latency of the server.

An important constraint was that the on-disk layout of the file system remain unchanged, so that users could migrate existing disks to the server without needing to back up and restore the file system. Finally, we wanted to restrict and isolate the changes to the file system, and avoid a large development effort or extensive code rewrite. These considerations precluded a log-structured file system approach.

An important decision was whether to use an *undo-redo* or a *redo-only* log [Moha 92]. An undo-redo log records, for each modified object, both its old and new values. The advantage of this method is that it has looser consistency requirements governing the order of log writes and in-place updates [Chut 92]. On the other hand, it doubles the size of the log and of each write to it. The recovery algorithm is also more complex, since logged transactions can be either replayed or rolled back. A redo-only log only records the new value of each modified object. This imposes stricter ordering of operations (described below), but has a simple recovery algorithm and smaller log size. We decided to adopt a redo-only log.

A related decision was to use physical block addressing, as opposed to logical block addressing or operations logging. Our log entries identify blocks by their physical disk locations, rather than by their logical names in the file system. This makes recovery simple, since the log contains the destination address of each item. For systems that use a logically addressed log, it is generally incorrect to replay the log from the beginning if the system crashes during recovery. Hence the recovery process must itself log its progress. This makes recovery slow and complex. The same problem occurs with systems that log operations rather than the new value of the data.

Our basic approach is similar to that in the Cedar file system [Hagm 87]. Instead of storing the

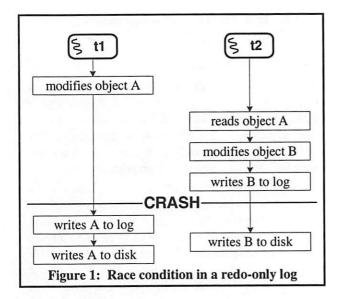
log in the same file system, however, we use a separate partition (preferably, a separate disk) for the log. This does not have to be a very large disk, since a metadata-only log does not take up too much space. Further, as we shall demonstrate, the log disk can be relatively slow, yet not degrade overall performance. This allows a system administrator to dedicate a small, inexpensive disk for the log. There is only one log in the system; it records changes to objects in all the file systems.

FFS has five different types of metadata objects – inodes, directories, allocation bitmaps, indirect blocks, and cylinder group summaries. We log modified inodes in their entirety. For directories, we record the 512-byte chunk that contains the modified data. In the case of allocation bitmaps, we only log the changed bits. Cylinder group summaries can be quickly computed from the allocation bitmaps, and hence are not logged. Logging indirect blocks would have required a substantial change to several functions, and the benefits are small, since this is a relatively infrequent operation. We decided to defer it in the first implementation.

The logging code only affects those NFS requests that can potentially modify the file system. We call these requests (setattr, write, link, symlink, create, remove, mkdir, rmdir, and rename) intrusive. Each intrusive request generates a log entry, which records all metadata changes made by that request. The server must write that entry to the disk before replying to the request. The in-place updates of the metadata wait for the next sync operation, which happens every 30 seconds in Calaveras.

When all metadata objects described by a log entry are *sync*'ed to disk, the entry is obsolete and can be overwritten. The log is circular, wrapping around when it reaches the end. If the log is large enough, the *sync* operations will keep it clean, and no separate garbage collection is necessary. We found that a 10-Mbyte log was sufficient for a server with 12 disks, each with about 375 Mbytes of active data, running at a load of 650 NFS operations per second. We therefore mandated a minimum log size of 32 Mbytes (way more than enough for the loads our server could support), and restricted garbage collection code to merely track the log usage, and to panic if active data was overwritten.

The initial performance results were extremely poor, due to the overhead of writing one log entry for each intrusive NFS operation. We devised a solution that allowed automatic batching of log entries without increasing the latency. We also added batching to the recovery algorithm. These



enhancements, described in the following sections, allowed us to meet our performance goals.

3.1. Log consistency

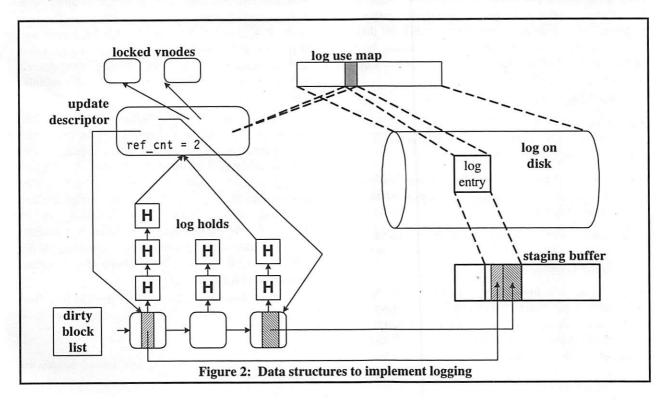
Crash recovery with a redo-only log is effected by replaying the log, writing all objects back to their correct locations on disk. This requires that the log copies of the objects are more current than the on-disk copies. Hence during normal operation, any change to a metadata object must be committed to the log before the in-place update of the object. The file system satisfies this condition by not releasing

the buffers containing the metadata objects to the cache until the log write completes.

There is, in fact, a much stronger requirement. It is incorrect to for an intrusive request to even read any object that has not been written out to the log. Figure 1 illustrates a potential problem. Thread t1 modifies object A, and is about to write it out, first to log and then to disk. Before it can do so, thread t2 reads object A, and based on that, modifies object B. It then writes B to the log, and is about to write it to the disk. If the system were to crash at this instant, the log contains the new value of B, but the new value of A is neither in the log nor on disk. Since the change to B depends on the change to A, this situation is potentially inconsistent.

To take a concrete example, suppose t1 is deleting a file from a directory, while t2 is creating a file by the same name in the same directory. t1 deletes the file name from block A of the directory. t2 finds that the directory does not have a file by that name, and proceeds to make a directory entry in block B of the directory. When the system recovers from the crash, it has the old block A and the new block B, both of which have a directory entry for the same file name.

To ensure consistency, the server must lock all metadata objects until their log entries are written out. This lock only affects intrusive NFS operations. It is perfectly valid to read uncommitted data if no



modifications are made based on that. A separate logging lock has been added to vnodes to implement this synchronization. Only the intrusive requests acquire this lock.

4. Implementation

Figure 2 describes the data structures used to implement logging. There are five main objects:

- an update descriptor tracks all operations and synchronization associated with a single NFS request.
- a *log entry* contains the actual data that is written to the log for one NFS request.
- the *log use map* tracks which parts of the log are active and which are free.
- staging buffers allow automatic batching of log writes.
- log holds prevent premature release of an update descriptor.

4.1. The update descriptor

An update descriptor baby-sits an NFS request, and holds all temporary information required to successfully complete it. Each intrusive NFS request first acquires an update descriptor, and passes it as an additional argument to many of the functions it calls. The descriptor is released not when the request completes, but when all the metadata objects that it has modified have been successfully updated in-place. At that time, the log entry on disk for this request becomes obsolete, and the corresponding bits in the log use map are cleared. This is the last step in processing an NFS request.

The fields of the update descriptor include

- a list of vnodes whose logging lock is held by this request. When the log entry is committed to disk, these locks are released.
- a reference count of the number of log holds pointing to it. The descriptor is released when this count drops to zero.
- base and size of the log entry. This determines the bits to clear in the log use map when the descriptor is released.
- a list of items to log. Each item is identified by an item type (inode, directory chunk, inode allocation bitmap, or block allocation bitmap), pointer to the cached data, an inode number, the address of the block on disk and the offset of the

object in that block.³ This information is used to create the log entry.

4.2. Log management

A log entry holds all the metadata changes for a single NFS request. It comprises a header and a table of contents, followed by the actual inodes, directory chunks, etc. The entry is padded to a 512-byte boundary. The table of contents describes the type and disk location of each object in the entry.

The header contains the number of items in the table of contents, as well as the total size of the entry. It has two other fields – recno and curhead. recno is a monotonically increasing record number, such that recno modulo the size of the log (in 512-byte sectors) equals the position of this entry in the log. curhead is the recno of the first active record in the log at the time the entry was generated. These fields are used for recovery. The tail of the log is the entry with the highest recno value. The curhead field of that entry identifies the head of the log. The recovery algorithm replays all entries between the head and the tail.

The log use map is a bitmap with one bit for each 512-byte sector of the log. When an entry is active, the bits for the corresponding sectors are checked. The bitmap allocates new entries to NFS requests, and frees them when their update descriptor is released. The map tracks the current head and tail, and returns the information along with each allocated entry. New entries must be allocated at the tail, since the log must always be contiguous. If the bits at the tail are busy (the tail wraps around and catches up with the head), the log is considered full, even if there are free regions in the middle. This results in a *panic*.

A log hold is simply a reference to an update descriptor. It contains pointers to chain the holds, and a pointer to the descriptor. In Calaveras, the file system maintains a dirty block list of modified metadata blocks; this list is periodically traversed and flushed by the *sync* daemon. The buffer headers for these blocks contain a linked list of log holds. Whenever an NFS request modifies a metadata object, it adds a log hold to the corresponding buffer, and increments the reference count of the update descriptor that the hold points to.

When *sync* successfully writes the block to disk, it traverses the list of log holds, releases each of

³For the allocation bitmaps, the item record contains the offset and value of the modified bits in the map.

them, and decrements the reference counts on the update descriptors. When the count reaches zero, the descriptor and its log entry are freed.

4.3. Normal operation

When an *nfsd* thread receives an intrusive request, it first allocates an update descriptor. It passes this descriptor to each vnode operation and onward to other functions that may need it. During the processing of the request, the thread acquires logging locks on the vnodes of any files or directories it accesses. Whenever it modifies a metadata object, it makes changes to the cached copy of the object. It does not release the corresponding buffer to the dirty block list, since it is not yet safe to write it to disk. It adds an entry in the update descriptor (in the list of items to log). This entry identifies the buffer and acts as a reference to it.

When all the processing for the request is complete, the thread calls the processUpdateDescriptor() routine, which performs the following tasks:

- Goes through the list of items to log, and computes the size of the log entry.
- Reserves disk space for the entry from the log use map.
- Reserves space in the staging buffer to write the entry.
- Traverses the list again, copying each item (inode, directory chunk, or allocation bits) to the staging buffer.
- Calls writeLog() to write the entry to disk.
- · Puts modified blocks on the dirty block list.
- Adds a log hold on each modified block, and increments the reference count on the

descriptor.

Releases all logging locks.

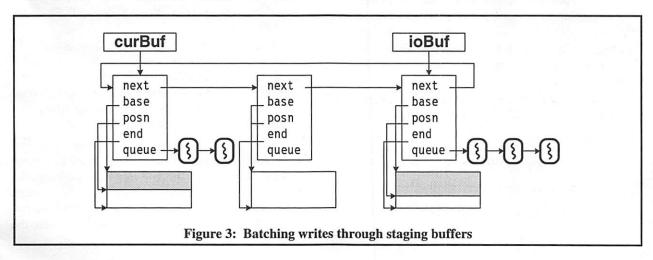
Multiple requests may modify a buffer between flushes. Each removes it from the dirty block list and replaces it after the log write completes. This results in multiple holds on each block. A block could also be modified more than once by a single request, for instance, when a request modifies two inodes in the same disk block. This causes multiple holds on the same block referencing the same descriptor.

Eventually, the *sync* daemon removes the block from the list and flushes it to disk. It then releases all holds for the block, and decrements the reference counts on the corresponding descriptors. If the reference count on a descriptor reaches zero, its log entry is marked obsolete (by clearing the bits in the log use map), and the descriptor is released as well.

5. Batching log writes

Writing each log entry to disk individually is expensive, and causes overall degradation of server performance. All logging systems rely on batching of log writes to obtain decent throughput. At the same time, the batching requires that some writes be delayed till sufficient data has been collected for a large write. Since NFS requests cannot be replied to until the write completes, this causes an increase in latency, which is usually unacceptable. For this reason, many logging file systems are unsuitable for an NFS server.

We devised a solution to this problem, to get automatic batching without any increase in latency. The basic principle is that under heavy load, the log disk should always be busy. No write should be kept waiting if the disk is idle; the batching is restricted to entries that accumulate while another log write is



in progress.

The staging buffers are used to create and populate the log entry before writing it to disk, and they also provide the mechanism for automatic batching. We need a minimum of two staging buffers, so that one is filled while the other is being written to disk. Normally, we use three buffers, linked on a circular list. The buffer size is large, since it must hold all the data transferred in a single log write. We use 64-Kbyte buffers, since that was the size of a single disk track. In practice, we never write more than 16 Kbytes at a time.

Figure 3 describes the organization of staging buffers. Each buffer header contains pointers to the *next* buffer, to the *start* and *end* of the data area, and to the first free byte in the buffer (*posn*). It also contains a queue of the threads that are blocked waiting for the buffer to be written to disk. There are two global pointers – *curBuf* points to the buffer currently being populated, and *ioBuf* points to the buffer currently being written to disk.

The processUpdateDescriptor() routine allocates the space needed for the log entry from the buffer pointed to by curBuf, starting at posn. If there is not enough space between posn and end for this entry, the next buffer becomes curBuf, and the entry is allocated from that. processUpdateDescriptor() then copies the metadata into the entry, and calls writeLog() to write the entry to disk.

writeLog() checks to see if another write is in progress (ioBuf is non-NULL). If not, it sets ioBuf to this buffer, advances curBuf to the next buffer, and initiates a write for this buffer. This will write out all log entries in this staging buffer. When the write completes, it wakes up all threads blocked on this buffer's queue.

If another write is already in progress, writeLog() adds the thread to the staging buffer's queue, and blocks till this buffer is written out. In this way, each time a log write completes, all pending writes are batched into a single I/O operation. The batching does not impose any additional latency.

This technique has some interesting properties. We define the average batching efficiency as

batching efficiency

- = log entries written / num of write ops
- = intrusive requests per second /

log writes per second

If the disk is constantly busy, the number of log disk writes per second is bounded by the raw performance characteristics of the disk, and is usually smaller due to the processing delays between writes. The efficiency then is the average number of intrusive requests that arrive in the time taken for one log write (including the associated processing plus idle time).

Under light loads, the efficiency is close to one, since there are few instances of multiple requests arriving in the before a disk write completes. As the incoming load increases, so does the batching efficiency. On a heavily loaded system, we measured efficiencies of up to 1.6. This translates to a 40% reduction in the total number of log writes.

The remarkable property of this implementation is that the speed of the log disk does not substantially impact the performance of the system. If the disk is slower, the number of log writes per second decreases, resulting in greater batching efficiency. Hence while a slow log disk may slightly increase the latency of each operation, the overall throughput is not impacted, and may in fact improve.

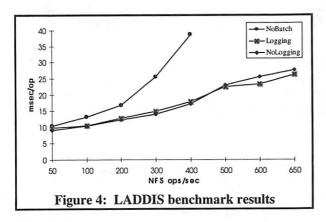
6. Log recovery

Recovering the log is conceptually straightforward – the log must be played back from start to end, writing back each metadata object in it to its in-place location. The two main implementation problems are how to find the start and end of the log, and how to reduce the total I/O needed for the recovery.

The log is divided into 64-Kbyte segments, whose only property is that log entries may not cross segment boundaries. The log allocation routines enforce this by padding the last entry of a segment to the boundary. Thus the beginning of a segment always coincides with the start of a new log entry. The entry's header contains its size, so the recovery algorithm can sequentially traverse entries in a segment.

The *recno* field in the entry is a monotonically increasing record number (such that *recno* % log size equals the position of the entry in the log). Hence the tail of the log is the entry with the highest *recno* value (the log may wrap around several times, but *recno* continues to increase).

The recovery procedure first uses a binary search of the segments (looking only at the first entry of each segment) to find the segment containing the highest *recno* (tail entry). It then reads the segment into memory, and scans it linearly to locate the tail entry. The *curhead* field in that entry identifies the



head of the log. It then recovers the log one segment at a time, starting at the head.

To recover a segment, it reads the whole segment into memory, and scans all entries to eliminate items obsoleted by later items in the same segment. For instance, several filenames may have been written to a directory chunk; the last instance of the chunk in the segment contains all the changes (to that point). It then sorts the remaining items based on file system id and block number. Finally, it copies the items back from the sorted list to the inplace disk locations. This way, it can combine all entries in a segment that modify the same block.

This algorithm yields substantial savings. For a 32-Mbyte log (512 64-Kbyte segments), the binary search locates the head segment in 9 reads, as opposed to 256 reads for a linear search). By eliminating duplicate items, and sorting and combining the rest, we reduce the number of writes by about a factor of 8 to 10 in typical cases.

7. Performance

We made our measurements on a DECpc 560ST, a 60 MHz Pentium machine with 192 Mbytes of RAM,

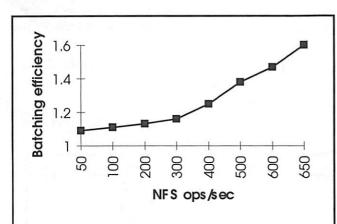


Figure 5: Batching efficiency as a function of server load

an EISA bus, a DEFEA FDDI controller, and two Adaptec 1740 controllers. For the logging benchmarks, we used 10 Digital RZ-26 disks for file systems, and one more for the log. For the nonlogging tests, we used all 11 disks for the file systems. We ran the Spec LADDIS benchmark [Witt 93], using DEC Alpha systems running OSF/1 as clients. We added code to the server to track the peak active log size and batching efficiency.

The primary motivation for logging was quick crash recovery, and to measure that, we powered off the system when running at peak load. At that time, the file system had about 3 Gbytes of data over 10 (logging case) or 11 (no-logging case) disks. In absence of logging, the file system was recovered by *fsck*. In the best case, this took about 450 seconds. Whenever *fsck* ran into a serious error, it had to be run interactively; this took much longer.

By comparison, the log recovery took between 3 and 14 seconds for the entire file system, depending on the size of the log at the time of the crash. This gain is somewhat exaggerated due to the fact that on our server, *fsck* recovers one disk at a time, while the log recovers all disks simultaneously. Even accounting for that, the log recovery would outperform *fsck* by a large factor.

Figure 4 describes the results of the LADDIS benchmark. The maximum throughput we could achieve was about 650 NFS operations per second, both with and without logging. Beyond that, the CPU became a bottleneck. At that peak load, the peak active log size was 10 Mbytes, and the batching efficiency was 1.6 (40% reduction in log writes).

The NoBatch curve describes the results of the logging implementation without batched writes. Besides uniformly high latency, the throughput peaks out at 400 NFS operations per second, since the log disk becomes a bottleneck.

The NoLogging and Logging curves are not too different. The average latency is a little higher with logging at low loads, but lower at high loads. This reflects the effect of the batching efficiency increasing with load. Figure 5 describes the variation of batching efficiency with incoming load.

A close look at the LADDIS test suite explains the similarity between the logging and no-logging results. 80% of the LADDIS requests are non-intrusive (lookup, getattr, read, readlink, and readdir), which are completely unaffected by logging. Another 16% are write and setattr requests, which typically modify only one metadata object (the inode), and are also relatively unaffected by logging

Test	CD	RD	CF	RF	Total
No Logging	370	151	339	121	981
Logging	123	66	264	89	542
Gain	3.01	2.28	1.28	1.36	1.81

Table 1: Small file benchmark results. The numbers are the elapsed time in seconds

(an in-place write is replaced by a log write). Only 4% of the load consists of requests that affect multiple metadata objects (*create*, *remove*, *rename*, *link*, *mkdir*, and *rmdir*); these are the ones that benefit most from logging.

To concentrate on these, we ran a "small file benchmark", in which we ran four types of tests on a server with eight disks. The tests were:

- **CD:** create 16000 directories; each disk has 20 directories with 100 sub-directories each.
- **RD:** remove the above tree (thus, 16,000 *rmdirs*).
- **CF:** copy a 1-Kbyte file 16000 times (same distribution as for the CD test).
- RF: remove the above files.

Table 1 shows the results of the benchmark. Logging is faster by a factor of 1.28 to 3.01 in the four tests. The actual improvement of server performance is even better, since the tests take into account client-side processing as well as other requests such as lookup and getattr that are necessary for these operations.

Table 2 shows measurements made on the server of the average elapsed time (in milliseconds) for the four metadata requests tested by the small file benchmark. The improvements are substantial; *mkdir*, in particular, is improved by more than a factor of six.

It has been difficult to compare these results with other NFS servers, for lack of clear criteria to base comparisons on. While many researchers [Hagm 87, Selt 93] have published performance measurements on log-structured or log-enhanced file systems, they have concentrated on benchmarks that deal with local access. Conversely, many others [Hitz 94, Jusz 94] have published measurements of

NFS performance using LADDIS and other tests. Some of these do not use logging at all, and the others do not have any measurements that explicitly isolate the effect of logging.

[Hitz 94] provides the closest point of reference. Its FAServer runs on the Intel 486 platform, and uses a log-structured file system in conjunction with NV-RAM. The configuration, therefore, is not too different from ours. Their published measurements, for an eight-system cluster, extrapolate to about 400 NFS ops/sec per server, at an average latency of under 15 miliseconds. We compare that to our benchmark on the i486, which peaks at 597 NFS ops/sec. with a latency of 29 miliseconds. The low latency of the FAServer is almost entirely due to their use of NV-RAM, which allows them to complete operations without waiting for disk I/O.

8. Conclusion

The logging enhancements met all their goals. There was a tremendous improvement in crash recovery time. Under a heavy, mixed, load, logging yielded a small gain in latency and equal throughput. For small file and directory operations, the gains were substantial (a factor of 1.50 to 6.30 improvement in server latency).

All this was achieved at a low cost. The entire implementation and testing took about twelve manweeks. We wanted to keep code changes to a minimum, to make it easy to integrate enhancements to the baseline code. We were able to do that; the logging enhancements modify the ufs code in five regards:

- The update descriptor is passed along as an additional argument to a number of vnode operations and ufs functions.
- 2. Intrusive operations acquire logging locks on

Operation	mkdir	rmdir	create	remove
No Logging	106.4	50.1	30.4	36.5
Logging	16.9	17.3	17.1	24.4
Gain	6.30	2.89	1.78	1.50

metadata objects they access.

- The functions that modify the metadata objects do not release the dirty buffers; instead, they add entries to the update descriptor's item list.
- 4. Intrusive requests allocate an update descriptor at the beginning and call processUpdateDescriptor() at the end.
- The sync() routine releases log holds and update descriptors.

Besides these changes, we had to write the logging module. This consisted of the functions and data structures that implement logging and recovery. The module has limited interaction with the baseline code, and is accessed through a narrow, well-defined, interface.

We left the FFS on-disk structures unchanged. This allows users to migrate existing disks to our server without backing up and restoring all files. The log needs to be kept on a separate disk, but that can be a small, inexpensive disk. The automatic batching technique gives greater savings under heavy load, and ensures that the performance of the log disk does not impact overall system throughput.

Some work remains to be done. We need to extend logging to include indirect blocks. We need to integrate performance improvements such as NFS write gathering [Jusz 94] and file system clustering [McVo 91] with our logging framework. We must add a checksum to log entries to guard against a crash leaving behind a partially written log entry. We also need to provide a background *fsck*-like function to recover from hard disk errors that corrupt one or more sectors. Finally, we must evaluate the suitability of the implementation for an NFS v3.0 server [Pawl 94]. The new version of the protocol, with its inherent ability to batch writes, might change the assumptions made in our design.

9. Acknowledgments

Several people have contributed to this effort. In particular, we would like to thank the rest of the Calaveras team – Percy Tzelnic, Steve Glaser, Lev Vaitzblit, Wayne Duso, and K. K. Ramakrishnan for their help and discussions throughout this project, and Chet Juszczak for his insightful discussions during the early presentations of this work.

References

[Chut 92] Chutani, S., Anderson, O.T., Kazar, M.L., Mason, W.A., and Sidebotham, R.N., "The Episode File System", Proceedings of the Winter 1992 Usenix Technical Conference, Jan. 1992, pp. 43-59.

[Finl 87] Finlayson, R.S., and Cheriton, D.R., "Log Files: An Extended File Service Exploiting Write-Once Storage", Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, Nov. 1987, pp. 139-148.

[Hagm 87] Hagman, R.B., "Reimplementing the Cedar File System Using Logging and Group Commit", Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, Nov. 1987, pp. 155-162.

[Hitz 94] Hitz, D., Lau, J., and Malcolm, M., "File System Design for an NFS File Server Appliance", Proceedings of the Winter 1994 Usenix Technical Conference, Jan. 1994, pp. 235-245.

[Jusz 89] Juszczak, C., "Improving the Performance and Correctness of an NFS Server", Proceedings of the Winter 1989 Usenix Technical Conference, Jan. 1989, pp. 53-63.

[Jusz 94] Juszczak, C., "Improving the Write Performance of an NFS Server", Proceedings of the Winter 1994 Usenix Technical Conference, Jan. 1994, pp. 247-259.

[Kaza 90] Kazar, M., Leverett, B., Anderson, O., Vasilis, A., Bottos, B., Chutani, S., Everhart, C., Mason, W.A., Tu, S., and Zayas, E., "Decorum File System Architectural Overview", Proceedings of the Summer 1990 Usenix Technical Conference, Jun 1990, pp. 151-164.

[Klei 86] Kleiman, S.R., "Vnodes: an Architecture for Multiple File System Types in Sun UNIX", Proceedings of the Summer 1986 Usenix Technical Conference, Jun 1986, pp.

[Kow 78] Kowalski, T., "FSCK: The UNIX System Check Program", Bell Laboratory, Murray Hill, NJ, Mar. 1978.

[McKu 84] McKusick, M.K., Joy, W.N., Leffler, S.J., and Fabry, R.S., "A Fast File System for UNIX", Transactions on Computer Systems, Volume 2, No. 3, Aug. 1984, pp. 181-197.

[McVo 91] McVoy, L.W., and Kleiman, S.R., "Extent-like Performance from a UNIX File System", Proceedings of the Winter 1991 Usenix Technical Conference, Jan. 1991, pp. 1-11.

[Moha 92] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwartz, P., "ARIES: A Transaction Recovery Method Supporting Fine-Grain Locking and Partial Rollbacks Using Write-Ahead Logging", ACM Transactions on Database Systems, Vol. 17, No. 1, Mar. 1992, pp. 96-162.

[Mora 90] Moran, J., Sandberg, R., Coleman, D., Kepecs, J., and Lyon, B., "Breaking Through the NFS Performance Barrier", Proceedings of the Spring 1990 European UNIX Users Group Conference, Apr 1990, pp. 199-206.

[Oust 89] Ousterhout, J.K., and Douglis, F., "Beating the I/O Bottleneck: A Case for Log-Structured File Systems", Operating Systems Review, 23(1), Jan. 1989, pp. 11-27.

[Oust 90] Ousterhout, J.K., "Why Aren't Operating Systems Getting Faster as Fast as Hardware?", Proceedings of the Summer 1990 Usenix Technical Conference, Jun 1990, pp.

[Pawl 94] Pawlowski, B., Juszczak, C., Staubach, P., Smith, C., Lebel, D., and Hitz, D., NFS Version 3 Design and Implementation", Proceedings of the Summer 1994 Usenix Technical Conference, Jun 1994, pp. 137-151.

[Rama 94] Ramakrishnan, K.K., Vaitzblit, L., Gray, C., Vahalia, U., Ting, D., Tzelnic, P., Glaser, S., and

Duso, W., "Operating System Support for a Video-On-Demand File Service", ACM Multimedia Journal, to appear.

[Rose 91] Rosenblum, M., and Ousterhout, J.K., "The Design and Implementation of a Log-Structured File System", Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles, Oct.

1991, pp. 1-15.

[Sand 85] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, R., "Design and Implementation of the Sun Network File System", Proceedings of the Summer 1985 Usenix Technical Conference, Jun 1985, pp. 119-130.

[Selt 93] Seltzer, M., Bostic, K., McKusick, M.K., and Staelin, C., "An Implementation of a Log-Structured File System for UNIX", Proceedings of the Winter 1993 Usenix Technical Conference, Jan. 1993, pp. 307-326.

[Thom 78] Thompson, K., "UNIX Implementation", Bell System Technical Journal, 57(6), Jul.-Aug. 1978.

[Witt 93] Wittle, M., and Keith, B., "LADDIS: The Next Generation in NFS File Server Benchmarking", Proceedings of the Summer 1993 Usenix Technical Conference, Jun 1993, pp. 111-128.

Uresh Vahalia received an MS in Computer Science from Syracuse University in 1985. He has since been working on operating systems and network protocols. He is currently at EMC Corporation, wehre he is developing file and continuous media servers. Uresh can be reached by email at vahalia@emc.com.

Dennis Ting received a BS in Computer Science from the University of Utah in 1971. He is currently a principal engineer at EMC Corporation. He has been involved in the development of real-time operating systems, network protocols, and high-performance network file servers.

Heuristic Cleaning Algorithms in Log-Structured File Systems

Trevor Blackwell, Jeffrey Harris, Margo Seltzer Harvard University

Abstract

Research results show that while Log-Structured File Systems (LFS) offer the potential for dramatically improved file system performance, the cleaner can seriously degrade performance, by as much as 40% in transaction processing workloads [9]. Our goal is to examine trace data from live file systems and use those to derive simple heuristics that will permit the cleaner to run without interfering with normal file access. Our results show that trivial heuristics perform very well, allowing 97% of all cleaning on the most heavily loaded system we studied to be done in the background.

1. Introduction

The Log-Structured File System is a novel disk storage system that performs all disk writes contiguously. Since this rids the system of seeks during writing, the potential performance of such a system is much greater than in the standard Fast File System [4], which must make writes to several different locations on the disk for common operations such as file creation. The mechanism used by LFS to provide sequential writing is to treat the disk as a log composed of a collection of large (one-half to one megabyte) segments, each of which is written sequentially. New and modified data are appended to the end of this log. Since this is an append-only system, all the segments in the file system eventually become full. However, as data are updated or deleted, blocks that reside in the log become replaced or removed and their space can be reclaimed. This reclamation of space, gathering the freed blocks into clean segments, is called cleaning and is a form of generational garbage collection [3]. The critical challenge for LFS in providing high performance is to keep cleaning overhead

low, and more importantly, to ensure that I/Os associated with cleaning do not interfere with normal file system activity.

There are three terms that will be useful in discussing LFS cleaner performance write cost, ondemand cleaning, and background cleaning. Rosenblum defines write cost as the average amount of time that the disk is busy per byte of new data written, including all the cleaning overheads [8]. A write cost of 1.0 is perfect meaning that data can be written at the full disk bandwidth and never touched again. A write cost of 10.0 means that writes are ultimately performed at one-tenth the maximum bandwidth. A write cost above 1.0 indicates that data had to be cleaned, that is, rewritten to another segment in order to reclaim space. Cleaning is performed for one of two reasons: either the file system becomes full, in which case cleaning is required before more data can be written, or the file system becomes lightly utilized and can be cleaned without adversely affecting normal activity. We call the first case when the cleaner is required to run, ondemand cleaning and the latter, optionally running the cleaner, background cleaning. Using these three terms, we can restate the challenge of LFS as minimizing write cost and avoiding on-demand cleaning.

The cleaning behavior of Sprite-LFS was monitored over a four-month period [8]. The measurements indicated that one-half of the segments cleaned during a four month period were empty and that the maximum write-cost observed in their environment was 1.6. While this write cost is acceptably low, the results do not give an indication of the impact on file system latency that resulted from cleaner I/Os interfering with user I/Os. Seltzer et al. report that cleaning overheads can be substantial, as

much as 41% in a transaction processing environment [9]. However, the cleaning cost in a benchmarking environment is an unrealistic indicator since the benchmark is constantly demanding use of the file system. Unlike benchmark environments, the realworld behavior of most workstation environments is observed to be bursty [1][5]. For example, consider an application that has two phases in which it executes. In phase 1, it creates and deletes many small files. In phase 2, it computes or uses the network, or terminates. Examples of such applications include Sendmail and NNTP servers. In FFS, the writes for the new data are all performed at the time the creates and deletes are issued. In LFS, all the small writes are bundled together into large, contiguous writes, so bandwidth utilization exceeds 50%, and approaches 100% as file size increases. The cleaner can run during the non-disk phase of the application when it does not interfere with application I/O. This workload is diagrammed in Figure 1.

The goal of this work is to investigate the real cost of cleaning, in terms of user-perceived latency. Using trace data gathered from three file servers, we have found that simple heuristics enable us to remove nearly all on-demand cleaning (specifically, on the most heavily loaded system, only 3.3% of the cleaning had to be done on-demand). The target operating environment of this study is a conventional UNIX-based network of workstations where files are stored on one or more shared file servers. These results do not necessarily hold for all environments;

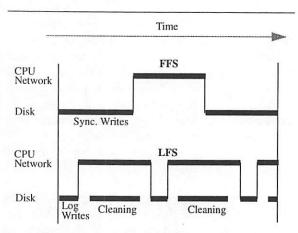


Figure 1. Overlapping cleaning with non-disk activity. The two traces depict FFS and LFS processing cycles. Both systems are writing the same amount of data, but FFS must issue its writes synchronously, resulting in a much longer total execution time. LFS issues its write asynchronously and cleans during processing periods, thus achieving much shorter total execution time.

they may not apply to online transaction processing environments where peak or near-peak transaction rates must be sustained at all times.

Section 2 describes the environments in which we gathered our measurements, Section 3 describes the traces we gathered, and Section 4 describes the tools and simulation model we used. Section 5 presents the results of our simulations, and Section 6 discusses the conclusions and ramifications of this work.

2. Benchmarking Methodology

All our trace data was collected by monitoring NFS packets to servers that provide only remote access. The NFSWatch utility [10], present on many UNIX systems, monitors all network traffic to an NFS file server and categorizes it, periodically displaying the number and percentage of packets received for each type of NFS operation. We modified the tool to output a log record for every NFS request. Since NFSWatch snoops on the ethernet, it does not capture local activity to a disk. To guarantee that we capture all activity to the file systems under study, we have limited our analysis to three Network Appliance FAServers [2], which are not general-purpose machines and provide no disk traffic other than serving NFS requests.

An issue in using Ethernet snooping is the potential for packet loss at the snooping host, which could skew the results. To detect such occurrences, we ran a daemon on a client host which referenced an unusually named file every 6 minutes. We could then search the log to find missing records. Only two lost requests out of 2200 were detected, indicating that the traces are fairly accurate. The lost packets were both at times of heavy demand, so while this form of loss may cause a slight underestimate of total traffic, it does not affect our estimates of idle time.

Two of the FAServers (Attic and Cellar) reside in Harvard's Division of Applied Science and one (Maytag) resides at Network Appliance's Corporate Headquarters. The Harvard environment consists of approximately 90 varied workstations (e.g. HP, DEC, SUN, X86) and X-terminals on two Ethernets. The DAS user community consists of approximately 100 users, most of whom use UNIX systems for text processing, email, news, software development, simulation, trace-gathering, and a variety of researchrelated activities. The Network Appliance environment consists of 90 workstations (mostly

Sparcstations,) X-terminals, PCs, and Macintoshes; the servers and some clients are on an FDDI ring. These machines service all business aspects of a small computer company: software development, quality assurance, manufacturing, MIS, marketing, and administration. The Network Appliance user community is approximately 60 users, of which 15-20 are heavy UNIX users, another 5-10 are casual users and the remaining use the PCs and Macintoshes nearly exclusively, placing little load on the NFS servers.

Machine Type	(Atti	rd DAS c and lar)	Network Appliance (Maytag)		
	Clients	Servers	Clients	Servers	
HP700	9	1	0	0	
Sun 3	0	2	0	0	
SparcStations	38	2	15	2	
DEC Alphas	5	1	0	0	
DEC MIPS	4	0	0	0	
FAServer	0	2	0	3	
x86 (DOS and UNIX)	6	1	17	5	
SGI	3	0	1	0	
Macintosh	1	0	31	0	
NeXT	5	0	0	0	
Router terminal concentrators	5	0	2	0	
X-Terminals	12	0	13	0	
Total	88	9	79	10	

Table 1. Measurement Environment. This table shows the per-machine-type breakdown of both the DAS and Network Appliance environments.

Table 1 summarizes our measurement environments. The client column indicates how many of the machines do not serve any files while the server column indicates how many of the machines act as NFS servers. The eight servers in the Harvard environment serve approximately 50 file systems. The nine servers in the Network Appliance environment serve approximately eleven file systems.

3. The Traces

We modified *NFSWatch* to log each operation, recording a timestamp, the sender's IP address, the server's IP address, packet size, request type, and request specific information. The request-specific information identifies the file being accessed, the offset in the file, and the number of bytes. The traces were gathered for 24 hours per day for several days (9 for Attic, 12 for Cellar, and 2.5 for Maytag). The Maytag traces would have been longer, but crashes of the tracing machine at Network Appliance prevented us from gathering a single, longer trace. Table 2 summarizes the file system activity during the trace period.

Characteristic	Attic	Cellar	Maytag
Total Requests	4.0 M	1.5 M	7.5 M
Data Read Requests	835 K	177 K	495 K
Reads from Cache	20.3 GB	8.9 GB	36.6 GB
Reads from Disk	7.1 GB	1.1 GB	3.4 GB
Data Write Requests	349 K	194 K	330 K
User Writes to Disk	3.4 GB	1.5 GB	5.5 GB
Directory Reads lookup, readdir	1.9 M	0.9 M	2.3 M
Directory Modifications create, mkdir, rm, rename, rmdir	43 K	13 K	410 K
Inode Updates	427 K	215 K	764 K
Trace Length	9 days	16 days	2.5 days
File System Size	4 GB	1 GB	8 GB
Disk Space Utilization	90%	90%	90%

Table 2. Summary of trace data collected on the three FAServers. All traces were gathered using the NFSWatch utility. We adjusted the disk sizes to achieve 90% utilization through most of the traces.

Because we gathered the trace data by capturing NFS requests, there are a few limitations in our data. The NFS_CREAT call returns an inode number (as part of the file handle) which is not captured by our trace gathering tools. In order to determine which inode was created in response to the NFS_CREAT call, we perform a look-ahead to the next NFS_WRITE or NFS_SETATTR call from the same client for the same file system and assume it belongs to the newly created inode. There are only two cases that are problematic: the creation of two or

more files in rapid succession by the same client and the creation of a file which is not immediately written, and whose attributes are not set. Both of these cases are detectable and a pass over our traces reveals that we correctly matched the NFS_CREAT with the inode number 99% of the time

NFS caching differs from local file system caching. Since NFS client caches timeout, we record requests at the server that would normally be satisfied by the local filesystem buffer cache. However, since we model a large (10 MB) cache on the server, this phenomenon does not affect results of disk behavior. Another artifact of NFS is that writes can arrive at the server in bursts corresponding to the number of biods running on the client. As will be discussed in Section 4, we model two different protocols for writing the NFS requests. These different models allow us to emulate both NFS and local file system behavior.

The shortcomings of using NFS as our trace-gathering method are outweighed by its advantages. By examining traffic on the local Ethernets, our tracing has no effect on the data gathered (our trace files are not written to the servers under examination). Other tracing methods that run on the server itself would not provide this unobtrusiveness.

4. The Simulator

We analyzed LFS behavior by simulating an LFS file system for each server. The simulator consists of approximately 4000 lines of ParcPlace Smalltalk in 55 classes. The simulator maintains a ten megabyte cache of in-memory blocks and maintains counts of the number of operations requested, the cache hit rate, the number and placement of I/Os in response to NFS requests, and the number of sectors read and written for on-demand and background cleaning. In addition, a graphical display depicts the file system during simulation, enabling the user to visualize file layout, data expiration, and cleaning. The simulator processes trace records at a rate of approximately 1000 records per second (half that if the graphical display is updated frequently). If desired, the simulation can be slowed sufficiently that every disk access can be observed graphically.

Since we are analyzing a live file system, we must create an equivalent LFS version of the file system as it exists before our tracing begins. We can then apply our trace data to this initial state. Since the file system under study is not an actual LFS, we must process the existing initial configuration and create an

LFS that realistically depicts the file layout one would expect on a LFS. The main point of interest in the initial state is how files are distributed across segments. In an ideal configuration, each file would be allocated contiguously on disk and all files smaller than a segment would reside within the same segment. In reality, files may be distributed across multiple segments because of the order in which writes occur. If writes to many different files are intermixed, then segments may contain parts of several files. In order to create a realistic initial state, we use the timing of NFS operations in the actual traces to provide an indicator of how much files should be interleaved in the initial state.

Before tracing, we snapshot the file system to record the size of every file. We group the files into bins based on the log₂ of the number of blocks in the file. We then process the traces to compute the average number of segments spanned by files of each bin size. Finally, we take each file in the initial configuration and distribute it evenly to the average number of segments spanned by files of its size. Within a segment, we allocate files sequentially. In this manner, we create an initial configuration consistent with the data gathered in our traces. To reduce the memory requirements of the simulation, files that are never referenced during the trace do not have space allocated for them on the disk.

The simulator provides a high-level model of LFS. It maintains file objects that record the location of their blocks, and directory objects that contain lists of the files they contain. The inode map, which maps inode numbers to the disk blocks in which they reside [7], is not directly simulated. Instead, each file is assessed 16 bytes of overhead in addition to the blocks that comprise it; this overhead is written as part of the segment summary.

The servers simulate a ten megabyte least-recently-used cache of data blocks, directory data, and inodes. If requested data are present in the cache, the request is serviced immediately and no disk operation is invoked. If the data are not present, then a disk request is scheduled.

The simulator reads each trace record and examines the operation type. If the record is a read operation (either data or meta-data), the simulator consults the appropriate object and determines if the required data are in the cache. If they are, the referenced data are made most-recently-used (MRU). If they are not, a disk request is recorded and the new

blocks are added to the cache. If the request is a write, then the data are added to the current segment, the old copy of the data (if it exists) is marked invalid in the in-memory representation of the disk map, and the simulator counters are updated. In a real LFS data are not explicitly marked invalid; instead the number of live bytes are maintained per segment and the cleaner detects invalid blocks by consulting the file meta-data (i.e. inode). Our explicit marking of data does not change the behavior of cleaning, it merely simplifies our implementation.

We simulate two write algorithms The first, called *NFS-async*, does not provide synchronous NFS semantics. It assumes that either the file systems are mounted with the unsafe NFS option [6] or that the system is equipped with at least two segments worth of non-volatile RAM into which segments are written before they are transferred to disk. In either case, the system caches data until a full segment accumulates before writing it to disk. The second model, *NFS-synch*, supports the synchronous behavior of NFS by writing a partial segment for each operation.

5. Results

Our results fall into three categories. First, we examine the behavior of the disk system, analyzing the frequency and length of idle intervals. The distribution of idle intervals indicates what heuristics will be effective for initiating the cleaner. Using these heuristics, we then examine how much data accumulates between cleaner invocations. This determines the maximum disk utilization that should be employed to avoid cleaner interference with normal disk activity. Finally, we examine the disk queue lengths that arise, giving an indication of the latency observed by the clients.

5.1. Idle Time Distribution

During the simulation, we recorded statistics on the length of idle intervals at the disk. For each of the servers under study, Figure 2 shows the distribution of intervals during which the disk was idle under the *NFS-async* model. We have removed all idle gaps of less than one second since these intervals predominate, but are sufficiently short that the cleaner will never run in them. The important feature of these distributions is that while most gaps were small (less than 1 second and not depicted in Figure 2) the absolute number of large intervals is sufficient to perform cleaning. More importantly, a long interval (greater than 2 seconds) is a good predictor of an even longer interval (greater than 4 seconds). For our file systems

using the asynchronous NFS algorithm, the probability that an idle period is at least four seconds long, once it is already two seconds long is more than 95% (96% on Attic, 97% on Cellar, and 98% on Maytag).

Because cleaner and user disk accesses are identified as such in our simulated disk queue, we were able to compute another measure of cleaning impact: the average number of cleaner writes in the queue when a user write was added to the queue. Table 3 shows the results, which indicate that interference is minimal - at most 0.07 requests.

System	Model	Interference
Attic	NFS-sync	0.069
	NFS-async	0.067
Cellar	NFS-sync	0.057
	NFS-async	0.072
Maytag	NFS-sync	0.039
	NFS-async	0.022

Table 3. Cleaner Interference. Interference numbers are the average number of cleaner requests in the disk queue when a user request is added. These numbers place an upper bound on the amount of cleaner interference with user activity. Cleaner activity writes to the same part of the disk as user activity, so often cleaner writes can be scheduled along with user writes without a major performance impact.

Figure 3 shows the same distributions as Figure 2, but for the *NFS-sync* model. In this case, each NFS request that changes the file system (create, write, mkdir, rm, rmdir, rename, setattr) is written to disk as a partial segment. This reduces the number of long idle gaps and also increases the need for cleaning since each partial segment introduces 512 bytes of overhead for at most eight kilobytes of data. Still, the probability that a two second interval is a good predictor for an interval greater than four seconds is still high (96% on Attic, 98% on Cellar, and 86% on Maytag). The gradual slopes in the cumulative distributions shown in Figure 2 emphasize this phenomena.

Despite the fact that Maytag seems to have the highest chance of cleaner activity being interrupted, the data in Table 3 shows that the interference is the least of the file systems. The cleaner on Maytag was generally able to clean less-utilized segments than on

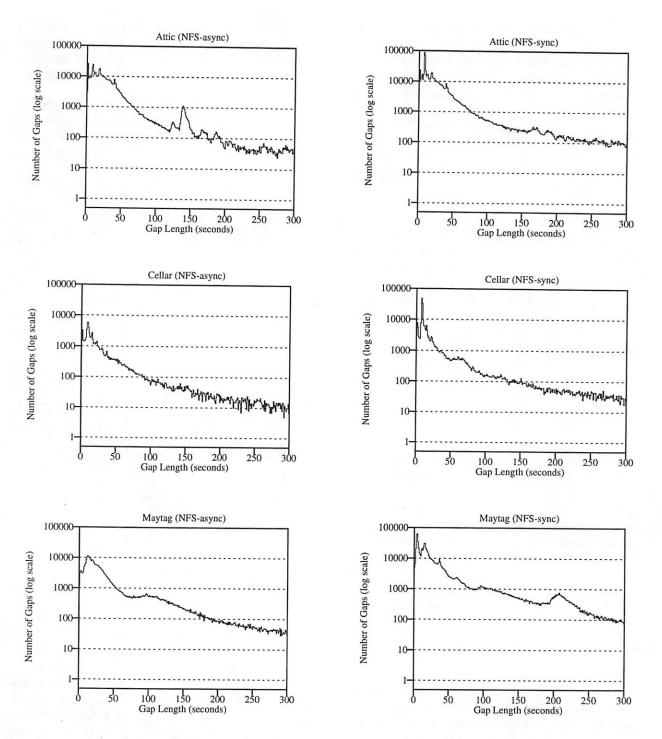
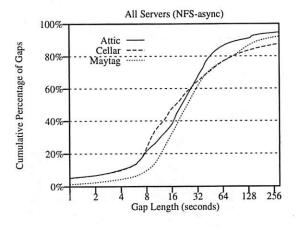


Figure 2. Idle Time Distributions for NFS-async. The three graphs show the distribution of idle disk intervals for each server. While most gaps are very short (the number of gaps less than 1 second completely dominates the distribution and has been omitted from these graphs), there are a sufficient number of large idle intervals that background cleaning can be performed frequently. The bump in Attic's trace at 120 seconds probably indicates a periodic process which often terminated an idle interval by performing some activity every 2 minutes.

Figure 3. Idle Time Distributions for NFS-sync. These distributions show similar behavior to those in Figure 2 but with a steeper curve. Since writes are sent to the disk immediately upon their arrival, there are more individual requests to the disk, the disk is used less efficiently and there are more shorter idle intervals. Even so, using a two second gap as a predictor is an effective heuristic for finding gaps that are large enough to permit cleaning of at least one segment.



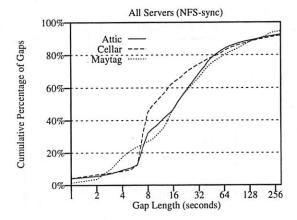


Figure 4. Cumulative Idle Time distributions. The gentle slope of the cumulative distributions between two and four second gaps emphasizes the predictive nature of two second gaps. Although most gaps are quite small (all gaps less than one second have been omitted from these distributions as is the case in the previous figures), there are still a sufficient number of large intervals in which to clean.

Attic or Cellar; the smaller average size of cleaner writes is responsible for the lower cleaner interference.

To understand how we can fit cleaning into these gaps, we need to quantify the time required to clean. In the best case, cleaning is free because there are segments that have no live data remaining in them (they are easily identified because we keep a count of the number of live bytes in each segment). In the general case, cleaning is summarized by the following algorithm.

- · Read a segment from disk.
- Determine which blocks are live.
- · Append the live blocks to the end of the log.

· Mark the segment clean.

The two long-running steps in this algorithm are the first, reading one megabyte from disk, and third, rewriting data into the log. In the pessimistic case, the read of the old segment will be a one megabyte sequential read, which takes well under a second on most of today's disks (e.g. a DSP 3501 SCSI disk can read 1 MB in approximately one-half second). Frequently, there will be few enough live bytes in the segment that we can avoid reading the entire segment and only read the blocks that need to be cleaned. The write time depends on the utilization of the segment being cleaned. If the segment utilization is very high, then we may need to write most of a megabyte; if the segment is lightly utilized, we can write only a few blocks. Even in the worst case, a one megabyte write takes only 0.75 seconds on the DSP 3501. In this worst case, we can clean at a rate of one segment per 1.25 seconds and in most cases, we can clean at a substantially higher rate. Therefore, if we use the two second idle time predictor, more than 90% of the time the cleaner will complete at least one segment's worth of cleaning before any user requests arrive.

Using this algorithm on our trace data, we found that we were able to use only background cleaning on Attic and Cellar (although disk utilization was 90% on all three servers, Attic and Cellar never ran sufficiently close to the file system capacity that on-demand cleaning was triggered). Maytag required occasional on-demand cleaning — 3.3% of the total number of segments cleaned were on-demand.

5.2. Dirty Data Accumulation

Figure 5 shows the distribution of dirty data accumulation between cleaning intervals in the NFS-async model and Figure 6 shows the distributions in the NFS-sync model. The cumulative distributions are shown in Figure 7.

On Maytag, the most heavily utilized system, the write volume due to user requests was about 2.2 GB per day. Running the cleaner to generate 2.2 GB of clean segments takes no more than 2750 seconds on the DSP 3501 disk - in practice, we estimate it to take about one third of this time on average. The large number of significant idle gaps during the day allowed cleaning to occur many times during the day, so that large quantities of unclean segments did not accumulate.

The cumulative distributions shown Figure 7 illustrate how rarely large amounts of data are written

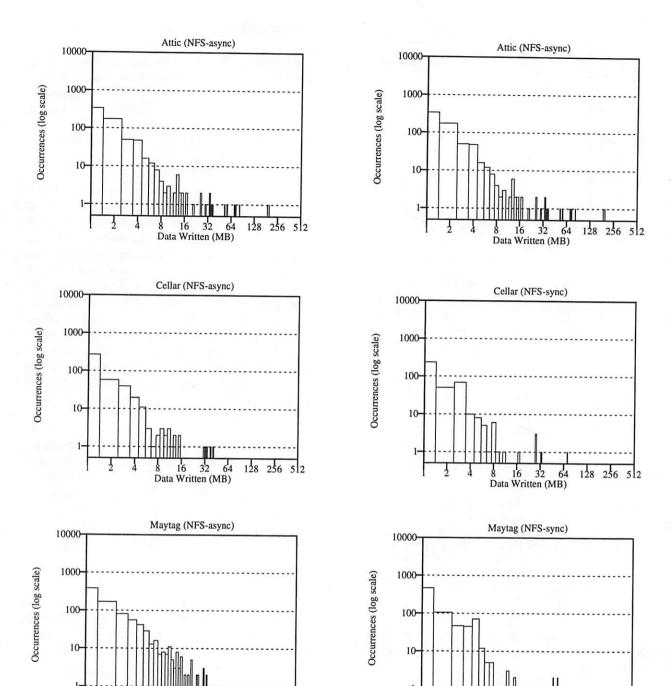
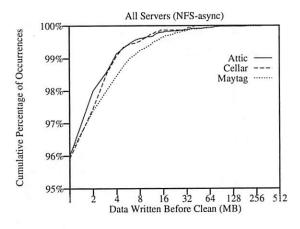


Figure 5. Distributions of Accumulated Dirty Data in the NFS-async model. Except for a few occurrences, the the amount of data written before cleaning was possible was small - less than 100 MB. On a multi-gigabyte file system that is limited to 90% fullness, there is plenty of space for accomodating this amount of data without requiring on-demand cleaning. We never observed more than 350 MB (4.5% of Maytag's disk space) written before cleaning occurred.

Figure 6. Distributions of Accumulated Dirty Data in the NFS-sync model. In the NFS-sync model, we never observed more than 420 MB (5.2% of Maytag's disk space) written before cleaning occurred.

128 256 512



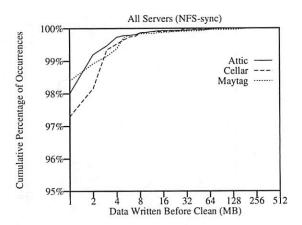
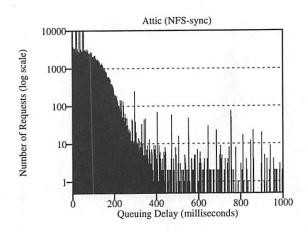


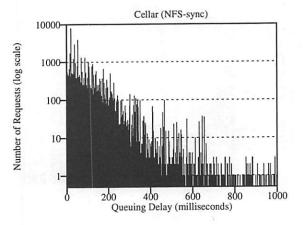
Figure 7. Cumulative Distributions of Dirty Data Accumulation. Note that the range of the graph starts at 95%. The vast majority (96%) of all write bursts are small (less than 1 MB). Despite the substantial difference in the average traffic levels to these machines, the distribution of many-megabyte bursts of write activity is similar.

before cleaning can be done. More than 96% of writes are less than one megabyte; 99% are less than four megabytes. This is true across all file systems studied.

5.3. Queueing Delays

The last area of investigation is the queuing delays observed at the disk. Log-structured file systems use the disk system efficiently by issuing large, sequential transfers. If the disk is being used efficiently, queueing delays should be kept to a minimum. However, large transfers also keep the disk busy for long bursts of time. Incoming requests that get queued after one of these long writes can wait for a long time. Figure 8 shows the queuing delays observed for the NFS-async model and Figure 9 shows the queuing delays for the NFS-sync model. The cumulative distributions are shown in Figure 10.





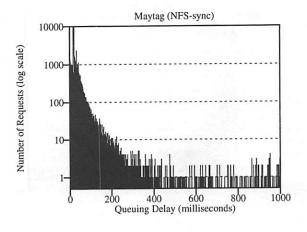
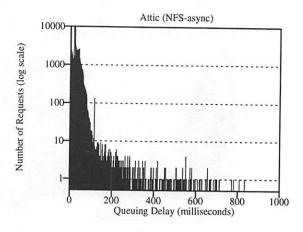
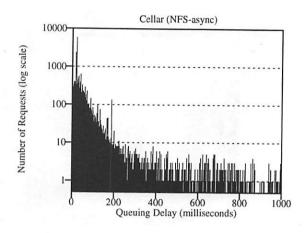


Figure 8. Distribution of Queueing Delays in the NFS-sync model. Comparing the two models, it is evident that delays are frequently longer in the NFS-sync case.





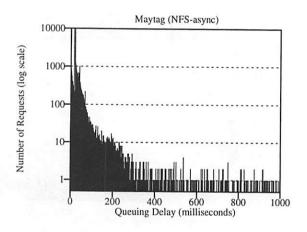
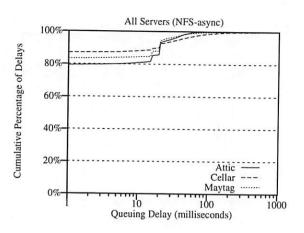


Figure 9. Distribution of Queueing Delays in the NFS-async model. Although there were occasional occurrences of large delays, most of the delays were small.



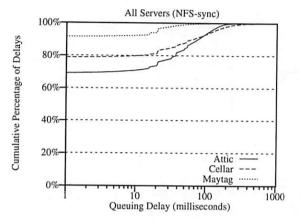


Figure 10. Cumulative Distributions for Queueing Delays. For all systems, more than 70% of the delays were zero, i.e. the request was put on an otherwise empty disk queue. Delays in the NFS-sync model were generally longer, probably due to both the increased write volume, and the fact that all writes for one request must complete before writes for another begin, which increases the number of rotational latencies incurred between writes.

All three file systems under both models could service the majority of requests with no queuing delay at all. Delays in the NFS-sync model were generally longer, due to both the increased write volume, and the fact that all writes for one request must complete before writes for another begin, which increases the number of rotational latencies incurred between writes.

Because our traces capture the time at which interactions happened between real clients and servers, the interval between closely spaced operations depends strongly on the speed of the real servers, as clients generally limit their maximum number of outstanding requests. To the extent that our

simulated file systems were faster or slower for various operations, the queuing delay may not be representative of a real system. However, the differences between NFS-sync and NFS-async reflect a real performance difference.

6. Conclusions

The simulation results are very encouraging for LFS. With a simple heuristic of cleaning whenever the disk has been idle for two seconds, we can virtually eliminate any user-perceived cleaning latency. Only a very small portion of disk queue latency is due to cleaner activity. Our results hold across two different environments: a university research environment and a commercial product development environment. However, some workloads (e.g. online transaction processing) may not demonstrate the idle-gap distribution on which this heuristic depends. In these cases, log-structured file systems must rely on other cleaning solutions [9].

7. Availability

The final trace set and the simulator software will be made available via Mosaic, at the URL: http://das-www.harvard.edu/users/students/
Trevor_Blackwell/Usenix95.html

8. Acknowledgments

We gratefully thank Network Appliance for providing us our first FAServer and for graciously running our tracing tools to enable us to provide trace data from very different environments. We also thank Hewlett Packard for the equipment grant that provided disk space on which to store our trace data and many, many CPU cycles for simulation. This work was funded in part by NSF grant CDA-94-01024.

9. References

- [1] Baker, M., Hartman., J., Kupfer., M., Shirriff, L., Ousterhout, J., "Measurements of a Distributed File System," *Proceedings of the 13th Symposium on Operating System Principles*, Monterey, CA, October 1991, 198-212.
- [2] Hitz, D., Lau, J., Malcolm, M., "File System Design for an NFS File Server Appliance," Proceedings of the 1994 Winter Usenix, San Francisco, CA, January 1994, pp. 235-246.

- [3] Lieberman, H., Hewitt, C., "A real-time garbage collector based on the lifetimes of objects," *Communications of the ACM*, 26, 6, 1983, 419-429.
- [4] McKusick, M.Joy, W., Leffler, S., Fabry, R. "A Fast File System for UNIX, "Transactions on Computer Systems 2, 3 (August 1984), pp 181-197.
- [5] Ousterhout, J., Costa, H., Harrison, D., Kunze, J., Kupfer, M., Thompson, J., "A Trace-Driven Analysis of the UNIX 4.2BSD File System," Proceedings of the Tenth Symposium on Operating System Principles, December 1985, 15-24.
- [6] Pawlowski, B., Juszczak, C., Staubach, P., Smith, C., Lebel, D., Hitz., D., "NFS Version 3: Design and Implementation," *Proceedings* of the 1994 Summer Usenix Conference, Boston, MA, June 1994, 137-152.
- [7] Rosenblum, M., Ousterhout, J., "The LFS Storage Manager," *Proceedings of the 1990 Summer Usenix*, Anaheim, CA, June 1990.
- [8] Rosenblum, M. and Ousterhout, J. "The Design and Implementation of a Log-Structured File System." ACM Transactions on Computer Systems, 10, 1 (February 1992), pp. 26-52.
- [9] Seltzer, M., Bostic, K., McKusick, M.K., and Staelin, C. "An Implementation of a Log-Structured File System for UNIX." Proceedings of the 1993 Winter USENIX Conference, January 1993, pp. 307-326.
- [10] SunOS System Administrator's Reference Manual, Section 8L.

Trevor L. Blackwell is a graduate student of Computer Science in the Division of Applied Sciences at Harvard University. His research interests include gigabit network design, efficient signalling protocol implementation, and wireless networking. He has worked at Bell-Northern Research since 1988, is currently supported at Harvard by BNR's External Research program. He was the recipient of a Canada Scholarship and IEEE McNaughton Scholarship. Blackwell received a B.Eng from Carleton University, Ottawa, in 1992. He can be reached at tlb@das.harvard.edu.

Jeffrey J. Harris is an undergraduate student of Computer Science in the Division of Applied Sciences at Harvard University. His research interests

include memory management systems, and the application of non-volatile memory to file systems. He has worked previously with Dr. Seltzer on a write-ahead file system. He is working towards his A.B degree in Computer Science from Harvard/Radcliffe College, expected in January of 1995. He can be reached at jjharris@das.harvard.edu.

Margo I. Seltzer is an Assistant Professor of Computer Science in the Division of Applied Sciences at Harvard University. Her research interests include file systems, databases, and transaction processing systems. She is the author of several widely-used software packages including database and transaction libraries and the 4.4BSD logstructured file system. Dr. Seltzer spent several years working at startup companies designing and implementing file systems and transaction processing software and designing microprocessors. She is a Sloan Foundation Fellow in Computer Science and was the recipient of the University of California Microelectronics Scholarship, The Radcliffe College Elizabeth Cary Agassiz Scholarship, and the John Harvard Scholarship. Dr. Seltzer received an A.B. degree in Applied Mathematics from Harvard/ Radcliffe College in 1983 and a Ph. D. in Computer Science from the University of California, Berkeley, 1992. She be reached can margo@das.harvard.edu.

ARCHITECTURE

Session Chair: Bob Gray, US WEST Technologies

NOTES

The New Jersey Machine-Code Toolkit

Norman Ramsey
Bell Communications Research
Mary F. Fernandez
Department of Computer Science, Princeton University

Abstract

The New Jersey Machine-Code Toolkit helps programmers write applications that process machine code. Applications that use the toolkit are written at an assembly-language level of abstraction, but they recognize and emit binary. Guided by a short instruction-set specification, the toolkit generates all the bitmanipulating code.

The toolkit's specification language uses four concepts: fields and tokens describe parts of instructions, patterns describe binary encodings of instructions or groups of instructions, and constructors map between the assembly-language and binary levels. These concepts are suitable for describing both CISC and RISC machines; we have written specifications for the MIPS R3000, SPARC, and Intel 486 instruction sets.

We have used the toolkit to help write two applications: a debugger and a linker. The toolkit generates efficient code; for example, the linker emits binary up to 15% faster than it emits assembly language, making it 1.7–2 times faster to produce an a.out directly than by using the assembler.

1 Introduction

The New Jersey Machine-Code Toolkit helps programmers write applications that process machine code—assemblers, disassemblers, code generators, tracers, profilers, and debuggers. The toolkit lets programmers encode and decode machine instructions symbolically. It transforms symbolic manipulations into bit manipulations, guided by a specification that defines mappings between symbolic and binary representations of instructions. We have written specifications for the MIPS R3000, SPARC, and Intel 486 instruction sets.

Traditional applications that process machine code include compilers, assemblers, linkers, and debuggers. Some applications avoid machine code by using assembly language; e.g., most Unix compilers emit assembly language, not object code.

Often, however, it is not practical to use an assembler, as when generating code at run time or when adding instrumentation after code generation. Applications that work on object code are more widely useful than those that require assembly code or source code, because they can be used on any executable file. Our toolkit makes such applications easier to implement.

Currently, applications that can't use an assembler implement encoding and decoding by hand. Different ad hoc techniques are used for different architectures. The task is not intellectually demanding, but it is error-prone; bit-manipulating code usually harbors lingering bugs. Our toolkit automates encoding and decoding, providing a single, reliable technique that can be used on a variety of architectures.

Applications use the toolkit for encoding, decoding, or both. For example, assemblers encode, disassemblers decode, and some profilers do both. All applications work with *streams* of instructions. Decoding applications use matching statements to read instructions from a stream and identify them, A matching statement is like a case statement, except its alternatives are labelled with patterns that match instructions or sequences of instructions. Encoding applications call C procedures generated by the toolkit. These procedures encode instructions and emit them into a stream; e.g., the SPARC call fnegs (r2, r7) emits the word 0x8fa000a2. Streams can take many forms; for example, a debugger can treat the text segment of a target process as an instruction stream.

The toolkit has three parts. The translator translates the matching statements in a C or Modula-3 program into ordinary code. The generator generates encoding and relocation procedures in C. The *library* implements both instruction streams and relocatable addresses, which refer to locations within the streams. The translator and generator need an instruction specification;

encoding procedures are generated from the specification, and matching statements can match the instructions or parts thereof defined in the specification. The library is machine-independent.

The toolkit's specification language is simple, and it is designed so that specifications resemble descriptions found in architecture manuals. It uses a single, bidirectional construct to describe both encoding and decoding, so their consistency is guaranteed. The toolkit checks specifications for unused constructs, underspecified instructions, and inconsistencies. An instruction set can be specified with modest effort; our MIPS, SPARC, and 486 specifications are 127, 193, and 460 lines.

We have two applications that use the toolkit. mld (Fernandez 1994), a retargetable, optimizing linker, uses the toolkit to encode instructions and emit executable files. ldb (Ramsey 1992; Ramsey and Hanson 1992), a retargetable debugger, uses the toolkit to decode instructions and to implement breakpoints.

The toolkit provides practical benefits, like reducing retargeting effort. For example, 1db's disassembler for the MIPS requires less than 100 lines of code, and mld has replaced 450 lines of hand-written MIPS code with generated encoding and relocation procedures. By hiding shift and mask operations, by replacing case statements with matching statements, and by checking specifications for consistency, the toolkit reduces the possibility of error. The toolkit can speed up applications that would otherwise have to generate assembly language instead of binary code. For example, mld emits executable files 1.7 to 2 times faster by using encoding procedures than by writing assembly language and using native assemblers. Such speedups would otherwise require hand-written encoding and relocation procedures for each target architecture.

The toolkit solves only part of the retargeting problem, but it solves that part completely. The solution is both elegant and practical; the toolkit's instruction-set specifications are clear, concise, and reusable, and the generated code is efficient. Our model of machine instructions makes several machine-level concepts general enough that they can be specified or implemented in a machine-independent way, including conditional assembly, span-dependent instructions, relocatable addresses, segments, object code, and relocation.

2 Using the toolkit

Figures 1 and 2 show how the toolkit is used in two applications. Code is shown in boxes, data in ovals. Code in doubled boxes is machine-dependent; a version exists for each target architecture. Code in single boxes is machine-

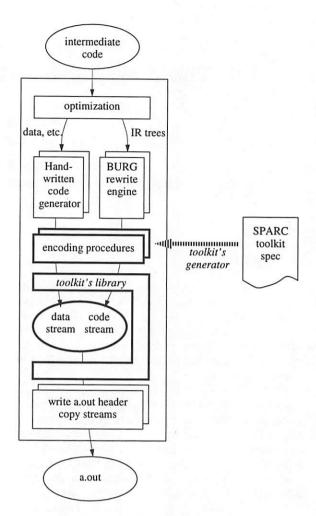


Figure 1: Structure of mld

independent. Code pointed to by thick, dashed arrows is generated by the toolkit. Boxes with heavy borders contain code that is part of the toolkit or generated by the toolkit. Ovals with heavy borders contain instruction streams that are written or read by toolkit-generated code. The names of the three parts of the toolkit are shown in italics.

mld

mld, shown in Figure 1, is a retargetable, optimizing linker for the MIPS and SPARC. mld links a machine-independent intermediate code, optimizes it, generates instructions and data, and emits a machine-dependent executable file (a.out). Retargeting mld requires adapting a code generator and writing code to emit an a.out file.

mld's code generators are based on those used in the lcc compiler (Fraser and Hanson 1991), which emit assembly code. Much of each lcc code generator is generated automatically from a BURG specification (Fraser, Henry, and Proebsting 1992), which rewrites intermediate-code subtrees to assembly-language templates; the rest of the code generator is written by hand. Adapting a code generator means modifying both the BURG specification and the hand-written parts. The toolkit simplifies those modifications. Code that prints assembly language is replaced with code that calls encoding procedures generated by the toolkit, e.g., the call

printf("fadds %%f1, %%f2, %%f3")

is replaced by the call fadds(f1, f2, f3), where fadds is generated by the toolkit. The encoding procedures use the toolkit's library to emit code into an instruction stream in mld's memory. The adapted code generators also call the library directly, e.g., to emit data.

mld uses instruction streams to model segments in an executable file. For example, a SPARC executable contains instructions in a "text" segment, initialized data in a "data" segment, and uninitialized data in a "bss" segment. mld uses one instruction stream for each segment. Instructions can be emitted into a stream even when they refer to labels whose positions are not yet known. The toolkit's generator creates code to resolve such references, so that mld needs only 20 lines of C code for relocation, and that code is machine-independent.

ldb

1db, shown in Figure 2, is a retargetable debugger for ANSI C. Most of its breakpoint implementation is machine-independent; the only machine-dependent part is the analysis of control flow (Ramsey 1994a). The analysis is written using a matching statement. The toolkit's translator can use any method of fetching instructions from streams; it works by instantiating templates that are supplied at translation time. For 1db, the templates fetch instructions from the program being debugged. The translator uses these templates and the SPARC instruction specification to transform the analysis's embedded matching statement into Modula-3 (Nelson 1991).

Matching statements make disassembly and flow analysis clear and concise. The analysis, for example, takes only a dozen lines of code; we present it in Section 9. Writing the analysis or disassembly by hand would require large nested case statements that enumerated all the SPARC opcodes.

3 Specification Concepts

Because machine instructions don't always fit in a machine word, the toolkit works with streams of instructions, not just the instructions themselves. An instruction stream is like a byte stream, except

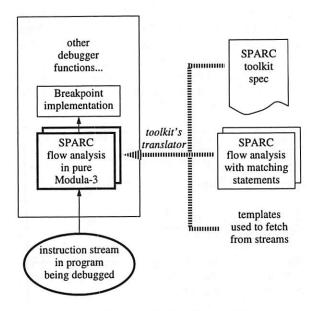


Figure 2: Use of decoding in 1db

that the units may be "tokens" of any size. An instruction is a sequence of one or more tokens; for example, a 486 instruction might include several 8-bit prefixes, an 8-bit opcode, 8-bit format bytes, and a 16-bit immediate operand.

Each token in an instruction is partitioned into fields; a field is a contiguous range of bits within a token. Fields contain opcodes, operands, modes, or other information. Patterns constrain the values of fields; they may constrain fields in a single token or in a sequence of tokens. Simple patterns can be used to specify opcodes. More complex patterns can be used for such tasks as specifying the structure of addressing modes or defining the group of 3-operand arithmetic instructions.

Constructors connect the symbolic and binary representations of instructions. At a symbolic level, an instruction is an opcode (the constructor) applied to a list of operands. The result of the application is a sequence of tokens, which is described by a pattern. Application programmers use constructors to emit instructions, by calling procedures derived from constructor specifications, and to decode instructions, by using constructors in matching statements to match instructions and extract their operands.

4 Tokens and fields

fields declarations specify how to divide tokens into fields. One fields declaration is given for each *class* of tokens; only fields named in the declaration can be extracted from tokens of that class. The declaration binds field names to bit ranges and specifies the number of bits in tokens of its

class. The toolkit generates the shifts and masks needed to get the value of a field in a token. Field values are always unsigned; a postfix exclamation point can be used to sign-extend them.

Architecture manuals have informal field specifications. For example, the fields for some SPARC load instructions are (SPARC 1992, p 90):

0	p		rd	0]	93	r	s1	i	33	simm13	
31	30	29	25	24	19	18	14	13	12	4-1	0

Other instructions may use a different format, e.g.,

op	rd			ор3	r	s1	ор	f		rs2
31 30	29	25	24	19	18	14	13	5	4	0

for floating-point arithmetic. This fields declaration defines the fields used in these and all other SPARC instructions:

The first two indented lines define the fields used in the formats pictured above; the last two lines define fields used in SPARC formats that aren't pictured in this paper. Because all SPARC instructions are 32 bits wide, only one class of tokens is needed, the 32-bit itoken (for "instruction token"). When instructions vary in size, more classes may be needed. On the Intel 486, instructions are composed of 8-, 16- and 32-bit tokens, which must be given different classes because they are of different sizes. It can even be useful to put tokens of the same size in different classes. For example, the 486 uses a "ModR/M" byte to specify addressing modes and an "SIB" byte to identify index registers (Intel 1990, page 26-3):

ModR/M	mod	reg/opcode	е	r/	m
SIB		5	3	2	0
	ss	index		ba	se
	7 6	5	3	2	0

The fields declarations for these bytes are:

Dividing tokens into classes helps detect errors in specifications. For example, putting the ModR/M and SIB tokens in different classes ensures that a user cannot mistakenly extract both a mod field and an index field from the same byte.

5 Patterns

Patterns constrain both the division of streams into tokens and the values of the fields in those tokens. When instructions are decoded, patterns in matching statements identify interesting inputs; for example, a pattern can be defined that matches any branch instruction. When instructions are encoded, patterns in the machine specification specify what tokens are appended to the stream.

Patterns are composed from *constraints* on fields. A constraint fixes the range of values a field may have. The typical range has a single value, e.g., op = 1. Patterns may be composed by conjunction (&), concatenation (;), or disjunction (|).

The basic patterns and pattern operators are best understood by considering how they affect matching. The constraint "lo <= f < hi" on a field f matches an input token if the f field of that token falls in the range defined by lo and hi. f is declared as a field of a particular class of tokens, which determines the size of the token matched. The wild-card constraint "some class" matches any token of class class, for example, on the SPARC, "some itoken" matches any 32-bit token. A conjunction "p & q" matches if both p and q match.\(^1\) A concatenation "p; q" matches if p matches an initial sequence of input tokens and q matches the following tokens. A disjunction "p | q" matches if either p or q matches.

Patterns in Specifications

The patterns declaration binds names to patterns. Pattern bindings are typically used to define opcodes. For example, the name call is bound to the pattern that corresponds to the SPARC opcode call by

The pattern op = 1 matches any 32-bit token in which bit 31 is zero and bit 30 is one. Opcodes can be defined by multiple constraints, for example

Defining opcodes individually would be tedious, and the result would be hard to compare with the architecture manual, which uses opcode tables. The patterns declaration can bind a list of names if a generating expression appears on the right. Generating expressions are modeled on expressions in the Icon programming language, which can produce more than one value (Griswold

¹Conjunction is permitted if and only if the constraints that are conjoined refer to fields in tokens of the same class; this restriction enforces the rule against mixing fields from different classes of tokens. For example, on the 486, the pattern mod = 0 & r_m = 5 is permitted, but the pattern mod = 0 & index = 2 is not.

and Griswold 1990). A generating expression is a pattern in which some integers have been replaced by expressions in brackets like {0 to 3}, which denotes the sequence of integers (0,1,2,3). These expressions are activated in left-to-right LIFO order, resulting in a list of patterns, each of which is bound to the corresponding name on the left. For example, the following declaration describes the first opcode table in the SPARC manual (SPARC 1992, p 227):

patterns

[TABLE_F2 call TABLE_F3 TABLE_F4]
is op = {0 to 3}

This definition binds the names TABLE_F2, call, TABLE_F3, and TABLE_F4 to the patterns op = 0, op = 1, op = 2, and op = 3, respectively. These names can now be used in the definitions of new patterns.

Most manuals give tables in which not every opcode is used. Unused opcodes can be bound to the special name "_", which may be used only on the left side of a binding. For example, Table F-3 from the SPARC manual defines many of the arithmetic opcodes (SPARC 1992, p 228):

		+	+	_	-	-	_
- 1	Jc	ıt	ı	e	Τ	ш	2

Pare	02110	1000			
	add	addcc	taddcc	wrxxx	
	and	andcc	tsubcc	wrpsr	
	or	orcc	taddcctv	wrwim	
	xor	xorcc	tsubcctv	wrtbr	
	sub	subcc	mulscc	fpop1	
	andn	andncc	sll	fpop2	
	orn	orncc	srl	cpop1	
	xnor	xnorcc	sra	cpop2	
	addx	addxcc	rdxxx	jmpl	
	_	_	rdpsr	rett	
	umul	umulcc	rdwim	ticc	
	smul	smulcc	rdtbr	flush	
	subx	subxcc	_	save	
		-	-	restor	е
	udiv	udivcc	w <u>r</u> "	2	
	sdiv	sdivcc		= 1: <u>==</u>]
			777	5000	21

TABLE_F3 & op3 = { 0 to 63 columns 4 }

The expression $\{0 \text{ to } 63 \text{ columns } 4\}$ generates the sequence $(0,16,32,48,1,17,33,\ldots,63)$, not the sequence $(0,1,2,\ldots,63)$, so that, for example, the name addcc is bound to the pattern op = 2 & op3 = 16. This trick makes it possible to use tables in which opcodes are numbered vertically.

6 Constructors

A constructor connects the symbolic and binary representations of an instruction by mapping a list of operands to a pattern. The toolkit's generator creates an encoding procedure for each constructor, so application writers can use constructors.

Constructors can also used within specifications; applying a constructor to a list of operands produces a pattern. Using constructors and patterns in each others' definitions helps organize the description of a machine's instruction set.

To reduce the possibility of errors, a specification writer can associate a type with a constructor. Applying a constructor of type T produces a value that is useful only as an operand to another constructor that expects an operand of type T. Applying an untyped constructor emits tokens into an instruction stream.

Because assembly language is the most familiar symbolic representation of instructions, we designed constructor specifications so their left-hand sides resemble assembly-language syntax: a constructor name and a list of operands. Operands may be separated by spaces, commas, brackets, or other punctuation. The punctuation is ignored; it is only syntactic sugar. The right-hand side of a constructor specification contains a pattern that describes the binary representation of the instruction specified. That pattern may contain free identifiers, which refer to the constructor's operands; such operands may be integers, or they may be patterns produced by constructors of a given type. For example, the following constructor describes the SPARC floating-point negate instruction:

constructors

fnegs n, m is fnegs & rs2 = n & rd = m

This definition of the *constructor* fnegs relies on a previous definition of the *pattern* fnegs, which appears on the right-hand side; that definition is

Using the name fnegs to refer both to a pattern and to a constructor may be confusing, but it is also desirable; architecture manuals normally use the same names in opcode tables and instruction descriptions. The toolkit's specification language makes the reuse possible by putting constructor names in a separate name space.

The specification of the constructor fnegs is not bad, but it is awkward to introduce integer operands n and m to refer to registers rs2 and rd. We can simplify by using *field operands* instead of integer operands.

constructors

fnegs rs2, rd is fnegs & rs2 & rd

On the right-hand side, the identifier rs2 stands for the pattern constraining the field rs2 to be equal to the first operand. This specification has fewer names to keep track of, but it has a new shortcoming: the same names appear in the same order on both sides of is, using only slightly different notation. This conjunction of all operands with the opcode is common in RISC machines, so

we provide a special abbreviation for it, in which the right-hand side is omitted:

```
constructors
fnegs rs2, rd
```

Because no constructor type is given, fnegs is untyped. The generated encoding procedure, which has the C declaration

```
void fnegs(unsigned rs2, unsigned rd);
```

has the side effect of emitting an fnegs instruction into the current instruction stream.

Not all operands are simple integers or fields. For example, the SPARC integer-arithmetic instructions take a second operand that may be a register or an immediate operand, depending on the value of the i field (SPARC 1992, p 84). Our SPARC specification defines the constructor type reg_or_imm for these operands, with register-mode and immediate-mode constructors:

```
constructors
```

where simm13! denotes the field simm13 interpreted as a signed integer. The identifier reg_or_imm can be used as an operand in the definitions of other constructors, for example

```
constructors
```

```
add rs1, reg_or_imm, rd
```

The encoding procedures generated from these declarations are:

The rmode and imode procedures have no side effects; they return values that can be passed to constructors like add.

Specifying similar constructors

unsigned rd);

Some architecture manuals describe instructions in alphabetical order; others group instructions with related syntax or semantics. The toolkit uses disjunction to define patterns that match any of a group of related instructions. These patterns can also be used to specify constructors. For example, the specification

patterns arith

```
is add | addcc | addx | addxcc | taddcc | sub | subcc | subx | subxcc | tsubcc | umul | smul | umulcc | smulcc | mulscc | udiv | sdiv | udivcc | sdivcc | save | restore | taddcctv | tsubcctv constructors arith rs1, reg_or_imm, rd
```

avoids repeated specifications for the constructors add, addcc, addx, and so on. When the constructor name on the left-hand side denotes a pattern, each disjunct of the pattern is used to generate a constructor. The patterns declaration attaches a name to each disjunct so that the constructor name can be determined.

Equations

Some instructions have integer operands that cannot be used directly as field values. The most common are PC-relative branches, in which the operand is the target address, but the corresponding field contains the difference between the target address and the program counter. Constructor specifications may include equations that express relationships between operands and fields. Equations contain sums of operand and field values with integer coefficients. For example, the specification for the SPARC branch instruction is

```
constructors
branch reloc { reloc = $pc + 4 * disp22! }
   is branch & disp22
```

The equation in braces shows the relationship between reloc, the target of the branch, \$pc, the program counter, and disp22!, the sign-extended displacement field. The toolkit detects that branch(reloc) is well-defined only if (reloc - \$pc) mod 4 = 0, and the generated encoding procedure enforces this restriction. We can introduce a machine-independent notion of program counter without a new specification concept; \$pc is simply a predefined identifier that denotes the place in the instruction stream where the constructor emits its tokens.

Some instructions use only part of an operand; for such instructions we can use *slices* of values; for example, val[10:31] denotes the most significant 22 bits of the 32-bit integer value val. This slice is used to specify the SPARC sethi instruction:

```
constructors
sethi val, rd { imm22 = val[10:31] }
  is sethi & rd & imm22
```

Equations may use inequalities as well as equalities. The toolkit does not use the inequalities to help solve the equations, but it does generate code which checks that the inequalities are satisfied.

Synthetic instructions and conditional assembly

Applying constructors in pattern definitions is most useful when defining "synthetic" instructions, i.e., instructions that are available in assembly language even though they are not part of the

```
constructors
                 : Eaddr
                                                is mod = 3 \& r_m = reg
 Reg
         reg
           [reg] : Eaddr { reg != 4, reg != 5 } is mod = 0 & r_m = reg
 Indir
 Disp8
                                                is mod = 1 & r_m = reg; i8 = d
          d[reg] : Eaddr { reg != 4 }
         d[reg] : Eaddr { reg != 4 }
                                                is mod = 2 \& r_m = reg; i32 = d
 Disp32
                                                is mod = 0 & r_m = 5;
 Abs32
                 : Eaddr
constructors
           [base][index * ss] : Eaddr { index != 4, base != 5 } is
 Index
                                                                               & ss
                                          mod = 0 & r m = 4: index & base
 Index8 d[base][index * ss] : Eaddr { index != 4 } is
                                          mod = 1 & r_m = 4; index & base
                                                                               & ss; i8
 Index32 d[base][index * ss] : Eaddr { index != 4 } is
                                          mod = 2 & r_m = 4; index & base
                                                                               & ss; i32 = d
                d[index * ss] : Eaddr { index != 4 } is
 ShortIndex
                                          mod = 0 & r_m = 4; index & base = 5 & ss; i32 = d
```

Figure 3: Constructor definitions for the 486's 32-bit addressing modes

real machine. For example, the synthetic instructions bset (bit set) and dec (decrement) are defined in terms of the real instructions or and sub (SPARC 1992, p 86):

```
constructors
bset reg_or_imm, rd is or(rd, reg_or_imm, rd)
dec val, rd is sub(rd, imode(val), rd)
imode is needed to convert the integer operand
val into an operand of type reg_or_imm.
```

Sometimes the best expansion for a synthetic instruction depends on the values of operands. We can choose one of several expansions by putting alternatives on the right-hand side of a constructor specification, each with its own set of equations. Each application of the constructor uses the first alternative for which the equations can be solved. For example, the SPARC synthetic instruction set has three ways to load a signed value val into register rd. When the 10 least significant bits of val are zero, use a single sethi instruction. When val fits in 13 bits, use an immediate-mode or instruction where the first operand is register 0, which is always zero. Otherwise, cut val into slices and use two instructions: sethi to assign the high-order bits and or to add the low-order bits:

7 CISC instructions

All MIPS and SPARC instructions can be specified by conjoining field constraints; this is the property that makes implicit right-hand sides useful. The 486 is not so simple. Both opcode and operands can span several tokens, and some tokens contain parts of each. Fields have multiple uses; for example the field r_m can indicate either a register choice or an alternate addressing mode, depending on its value. Figure 3 shows constructor specifications for the 486's addressing modes, illustrating how the toolkit's specification language handles CISC. The brackets and asterisks are not meaningful; they simply help show how the constructors correspond to standard assembly language.

Effective addresses contain a ModR/M byte, which contains an addressing mode and a register. In indexed modes, the ModR/M byte is followed by an SIB byte, which holds index and base registers and a scale factor. Finally, some modes take immediate displacements (Intel 1990, Tables 26-2 to 26-4). We define constructors of type Eaddr to create effective addresses in 32-bit mode. The first group of constructors specifies the non-indexed addressing modes. The simplest mode is encoded by mod = 3; it is a register-direct mode that can refer to any of the machine's eight general registers. The next three modes are register-indirect modes with no displacement, 8-bit displacement, and 32-bit displacement. The fields mod and r_m of the ModR/M byte are defined above; the fields i8 and i32 occupy full 8-bit and 32-bit tokens and are used to hold displacements:

```
fields of I8 (8) i8 0:7 fields of I32 (32) i32 0:31
```

Semicolons separate ModR/M bytes from the displacements that follow. The inequality reg != 5 shows that r_m may not take the value 5 in simple indirect mode. Instead of denoting indirect use of the base pointer, which is the register normally en-

coded by 5, the combination mod = 0 & r_m = 5 encodes a 32-bit absolute mode. The inequality reg != 4 in the equations associated with the register-indirect modes shows that the value 4 may not be used to encode indirect use of the stack pointer, which is the register normally encoded by 4. This value is used instead to encode the indexed modes, which use an SIB byte as well as the ModR/M byte.

The indexed modes are the second group in Figure 3. The ModR/M byte in which $r_m = 4$ is followed by an SIB byte. The stack pointer may not be used as an index register (index != 4). Depending on the value of mod in the ModR/M byte, the SIB byte may end the address, or an 8-bit or 32-bit displacement may follow. Finally, "mod = 0 & base = 5" denotes an indexed address with no base register and a 32-bit displacement.

8 Relocation

Instructions often refer to the addresses of data or of other instructions; e.g., to load the value of a variable or to branch to a label. The toolkit can emit such instructions even before the addresses are known. Instructions and data are emitted into relocatable blocks, which implement the instruction-stream abstraction. An application can write into a relocatable block without knowing where in memory the block's contents will eventually be located. Relocation assigns an address to a relocatable block. Applications may use any number of relocatable blocks.

A label points to a location in a relocatable block. The toolkit does not associate names with labels; applications can use any method they want to find and identify labels. A relocatable address is any quantity whose value depends on the value of a label. The toolkit's generator treats relocatable addresses as values of an abstract data type. The toolkit's library provides a particular implementation: to the library, a relocatable address is the sum of a label and a signed offset. This simple form is adequate for applications like compilers and linkers. Authors of more sophisticated applications can use more sophisticated representations (e.g., linear expressions over addresses and labels) without changing the toolkit.

Constructor operands whose names begin with reloc are relocatable addresses, like the reloc operand of the branch constructor shown above. The relocatable address \$pc can be used in a constructor's equations, where it denotes the location at which the constructor's tokens begin. When a constructor that uses relocatable addresses is applied, it checks to see if those addresses are known

(i.e., they have been assigned absolute addresses). If so, it treats them as ordinary integers and emits the instruction. Otherwise, it emits placeholder tokens and creates a relocation closure. The closure contains references to the unknown addresses, plus a pointer to a function that, when applied, overwrites the placeholder with the correct instruction. The application keeps the closure until the addresses it depends on become known, at which point it can apply the closure function and discard the closure.

For flexibility, we let applications decide how to organize relocation closures, when to apply them, and when to discard them. For example, a standard linker might store all closures in a simple list and discard them after applying them, because the absolute addresses of segments don't change after they are assigned. An incremental linker would keep the closures, because some might have to be re-applied when relocatable blocks were moved. It might store the closures in a more complex data structure, to avoid re-applying all closures when only a few relocatable blocks moved.

For placeholders to be computable, the specification writer must associate a placeholder pattern with each class of tokens. The toolkit uses the shape of a constructor's pattern to compute a placeholder for it, ensuring that the placeholder has the same shape as the instruction that overwrites it when the closure is applied. Placeholders are typically chosen so that attempts to execute them are detected. For example, we chose

placeholder for itoken is unimp & imm22 = Oxbad as the placeholder for the SPARC. A dynamic linker might use a special trap instruction as a placeholder; it could handle the special traps by resolving the unknown address and applying the instruction's closure at run time.

When a conditionally assembled constructor is applied to a relocatable address, it may not be possible to determine which sets of equations can be satisfied, because the value of the relocatable address may not be known. In that case, the toolkit makes the most conservative decision, choosing the first alternative whose equations are known to be satisfied. This technique, while safe, is not suitable for emitting span-dependent instructions; for example, it uses the most general representation for all forward branches.

9 Matching statements

Decoding applications use the toolkit's matching statements. These resemble ordinary case statements, but their arms are labeled with patterns. The first arm whose pattern matches is executed. Free identifiers used in these patterns are binding instances; they are bound either to field values or to the locations of sub-patterns within the pattern. For example, the matching statement

```
match p to
| fnegs & rs2 = i & rd = j =>
    printf("let f%d = - f%d", j, i);
| some itoken =>
endmatch
```

prints a message if the instruction pointed to by p is a floating-point negate, and it does nothing otherwise. The pattern some itoken always matches, so if the two arms of this matching statement were reversed, the toolkit's translator would issue a warning that the second arm could never be executed.

Just as in specifications, it is often more convenient to write patterns in the form of constructor applications, e.g., fnegs(i, j), in which the operands i and j are bound to integers by the matching statement. Pattern-valued operands are bound not to integers but to locations in the instruction stream; for example, the SPARC pattern add(0, rand2, rd) matches any add instruction in which rs1 is zero. rd is bound to the number of the destination register, and rand2 is bound to the location of the token containing the second operand.

Most applications decode streams of instructions in sequence, so they need to find the address of the following instruction as part of decoding the current instruction. The matching statement contains a special syntax for this common operation; if the word match is followed by an identifier in square brackets, the matching statement assigns the address of the first unmatched token to that identifier.

The translator implements the matching statement by building a decision tree. Each node of the tree tests one field of one token of the stream being matched. Tokens are identified by width and by offset from the location being matched; for example, the 32-bit displacement in an index-mode add instruction on the 486 might be "the 32-bit token at an offset of 24 bits." The translator first rewrites all patterns so that fields are tagged with the offset of the token to which they belong. It then attempts to build a decision tree that identifies the matching arm using a minimal number of nodes. No polynomial-time algorithm is known, so we use several heuristics. When we examine the trees produced by these heuristics, we see that they are as good as what we would write by hand.

Application writers can use any representation of instruction streams. They specify the representation by supplying the toolkit with four code fragments: the data type used to represent locations, a template used to add an integer offset to a location, a template used to convert a location to a program counter (an unsigned integer), and a template used to fetch a token of a specified width from a location. Widths are measured in bits; offsets are measured in the same units used for the program counter, which defaults to 8 bits per addressing unit. The application writer must supply code that can fetch tokens using the proper byte order, which is usually the byte order of the machine the application runs on.

Matching statements in 1db

1db, a retargetable debugger, uses matching statements to disassemble machine code and to help implement breakpoints. 1db is written in Modula-3. It uses an object type to represent an instruction stream of a program being debugged, and it uses unsigned integers to refer to locations in such streams. Here are the code fragments that give the toolkit's translator the representation of streams:

```
address type is "Word.T"
address add using "Word.Plus(%a, %o)"
address to pc using "%a"
fetch any using "FetchAbs(m, %a, Type.I%w).n"
```

The quoted strings are fragments of Modula-3 code in which %a stands for an address or location, %o stands for an offset, and %w stands for a width. Offsets and widths are measured in bits. Word.Plus is an unsigned add. The m argument to FetchAbs is an object representing the address space being debugged; it must be defined by the context in which matching statements appear.

Figure 4 shows a simplified version of the SPARC code in 1db's breakpoint implementation, omitting subtleties associated with delayed branches. This code finds which instructions could be executed immediately after an instruction at which a breakpoint has been planted (Ramsey 1994a). After an ordinary instruction, the only instruction that can follow is its inline successor, as computed by the first arm of the matching statement. FollowSet.T{succ} is a set of addresses containing the single element succ. Calls and unconditional branches also have only one instruction in their "follow set," but conditional branches have two. The two jmpl patterns are indirect jumps through registers; the GetReg procedure gets the value in the register in order to compute the target address. Using the matching statement implemented by the toolkit makes it clear what the code is doing; the logic would be obscured if implemented by nested case statements.

10 Implementation

The toolkit's generator and translator are 6000 lines of Icon (Griswold and Griswold 1990). The

```
PROCEDURE Follow(m:Memory.T; pc:Word.T):FollowSet.T =
VAR succ : Word.T;
BEGIN
  match [succ] pc to
  nonbranch
                                      => RETURN FollowSet.T{succ};
    call(target)
                                      => RETURN FollowSet.T{target};
  | branch(target) & (ba | fba | cba) => RETURN FollowSet.T{target};
  | branch(target)
                                      => RETURN FollowSet.T{succ, target};
   jmpl(dispA(rs1, simm13), rd)
                                      => RETURN FollowSet.T{GetReg(m, rs1)+simm13};
  | jmpl(indexA(rs1, rs2), rd)
                                      => RETURN FollowSet.T{GetReg(m, rs1)+GetReg(m, rs2)};
  some itoken
                                      => Error.Fail("unrecognized instruction");
  endmatch
END Follow;
```

Figure 4: Matching statement used for control-flow analysis of SPARC instructions

Machine	Spec	Ops	Insts	Modes	Time
MIPS	127	113	158	1	27
SPARC	193	184	260	2/4	102
Intel 486	460	496	623	8	635

Spec	Lines of specification
Ops	Opcodes in the manual's tables
Insts	Instructions specified
Modes	Address modes in most instructions
Time	Seconds to create encoding procedures
	(elapsed time on a SPARCstation 10)

Table 1: Characteristics of three machines

library is 600 lines of ANSI C. Table 1 shows some characteristics of our three machine specifications, including the amount of time needed to generate a complete set of encoding procedures. The number of addressing modes affects that time, because each encoding procedure has an alternative for each mode of each operand. Even the long generator time for the 486 is acceptable, because one rarely writes a specification containing hundreds of new constructors. (One can add a few constructors to an existing specification in time proportional to the number of added constructors, not to the size of the whole specification.)

The translator takes 10 seconds to transform either 1db's SPARC follow-set matching statement (Figure 4) or the analogous MIPS statement into Modula-3 code. The translator time, although shorter than the generator times, is more problematic, because the translator must be run after every change to a source file with matching statement.

The toolkit generates efficient code. mld, our example encoding application, can use the toolkit to emit binary, or it can emit assembly code. It

	MIPS			SPARC		
Program	Bin out	Asm +			Asm +	
eqntott	4.1	4.3 +	7.1			
li	7.2	7.5 +	14.7	7.9	8.2 +	5.0
espresso	14.7	17.5 +	33.0	14.2	15.4 +	11.4
					61.7 +	

Bin out	Use toolkit to emit binary a.out
Asm out	Emit assembly code, without toolkit
Run as	Translate assembly code to a.out

Table 2: Seconds to generate code & make a.out

always executes faster when emitting binary. For example, when linking and emitting binary code for the integer SPEC benchmarks, mld is up to 15% faster than when it emits assembly code, as shown in Table 2. Moreover, emitting assembly requires running the assembler, which increases the total time required to generate an a.out without using the toolkit: 1.7–2.1 times longer on the SPARC and 2.8–5.6 times longer on the MIPS. This comparison is unfair to the MIPS assembler, because the MIPS assembler schedules instructions but the toolkit does not.

Application writers can trade safety for more efficiency. By default, the toolkit checks the widths of field values, calling a user-defined error procedure if they overflow. Application writers unwilling to pay for a compare and branch can direct that field values silently be narrowed to fit. Those unwilling to pay even the cost of masking out highorder bits can assert that certain fields never overflow, in which case the values are used without masking. This choice is appropriate in some situations, for example, when field values denote registers and are chosen by a register allocator.

We measured encoding costs on a DEC 5000/240 with a memory-mapped clock. Simple encoding procedures like nop and mov cost less than 30 cycles when generated without safety checks; 6 of these cycles are for procedure call and return. Safety checks add 2 cycles per operand checked. Encoding a branch instruction, which requires checking relocatable addresses and doing a relative-address computation, costs 118 cycles.

11 Related work

Ferguson (1966) describes the "meta-assembler," which creates assemblers for new architectures. It works not from a declarative machine description but from macros that pack fields into words and emit them; it is essentially a macro processor with bit-manipulation operators and special support for different integer representations.

Wick (1975) describes a tool that generates assemblers based on descriptions written in a modified form of ISP (Bell and Newell 1971). His work investigates a different part of the design space; his machine descriptions are complex and comprehensive. For example, they describe machine organization (e.g., registers) and instruction semantics as well as instruction encoding. We prefer to build applications by using several simple specifications, each describing different properties of the same machine, to build different parts.

The GNU assembler provides assembly and disassembly for many targets, but different techniques have been applied ad hoc to support different architectures (Elsner, Fenlason, et al. 1993). For example, 486 instructions are recognized by hand-written C code, but MIPS instructions are recognized by selecting a mask and a sample from a table, applying the mask to the word in question, then comparing the result against the sample. On both targets, operands are recognized by short programs written for abstract machines, but a different abstract machine is used for each target. Another set of abstract machines is used to encode instructions during assembly. The implementations of the abstract machines contain magic numbers and hand-written bit operations. The programs interpreted by the abstract machines are represented as strings, and they appear to have been written by hand.

In spirit, our work is like ASN.1 (ISO 1987), which is used to create symbolic descriptions of messages in network protocols, but there are many differences. ASN.1 data can be encoded in more than one way, and in principle, writers of ASN.1 specifications are uninterested in the details of the encoding. ASN.1 encodings are byte-level, not bit-level encodings; ASN.1 contains an "escape hatch" (OCTET STRING) for strings of bytes in which

individual bits may represent different values. Finally, ASN.1 is far more complex than our language; for example, it contains constructs that represent structured values like sequences, records, and unions, that describe optional, default, or required elements of messages, and that distinguish between tagged and "implicit" encodings of data.

12 Future work

The names of instructions may conflict with names that application writers use in their programs; generating encoding procedures with those same names can cause name-space collisions. Different languages offer different solutions to this problem, e.g., C++ classes or Modula-3 interfaces. Plausible solutions for C include attaching a unique prefix to the names of all encoding procedures, or using a structure containing pointers to the encoding procedures. The latter choice costs more at run time, but could enable multiple sets of encoding procedures in the same application, e.g., one to emit binary and one to emit ASCII (for debugging).

It would be instructive to experiment with multi-pass strategies for conditional assembly, to make conditional assembly an efficient, machine-independent way of specifying how to assemble span-dependent branches. Such strategies can change the size of previously emitted instructions. Size changes could be accommodated by putting each varying instruction in its own relocatable block, but it would be awkward to expose these extra relocatable blocks to an application.

One could store relocatable blocks and relocation closures in a file and use the collection as a machine-independent representation of object code. Such an object-code format could make it easier to write testing tools like Purify (Hastings and Joyce 1992), profilers and tracers like qpt (Ball and Larus 1992), and optimizing linkers like OM (Svrivastava and Wall 1993), all of which manipulate object code. One could add a machine-independent linker to the toolkit's library and extend the generator to generate assemblers from specifications, making it possible to take assembly code generated by existing compilers and assemble it into this new format. The hard part of this approach would be externalizing the function pointers contained in the relocation closures.

We are designing an extension to the toolkit that enables it to describe arbitrary sequences. Such sequences are common components of network messages, and this extension would make it possible to use the toolkit to generate encoding and decoding code for such messages. We are also investigating the use of toolkit specifications to help build compression models for arithmetic coding (Bell, Cleary, and Witten 1990). It may be increasingly useful to compress machine code as the gap between processor speeds and secondary storage widens. Better compression would help reduce the storage requirements for large network traces, which can consume gigabytes (Duffy et al. 1994).

13 Discussion

Our specification language evolved from a simpler language used to recognize RISC instructions in a retargetable debugger (Ramsey 1992, Appendix B). That language had field constraints and patterns built with conjunction and disjunction, but no concatenation and no constructors. There was no notion of instruction stream; instructions were values that fit into a machine word. We extended that language to specify encoding procedures by writing a constructor name and a list of field operands to be conjoined. This scheme sufficed to describe all of the MIPS and most of the SPARC, and we used it to generate encoding procedures for mld. There are a few parts of the SPARC it could not describe, however, and it was completely unable to describe the 486, even with the addition of concatenation to the pattern operators. Two changes solved all our problems: making patterns explicit on the right-hand sides of constructor specifications, and using constructor types to permit patterns as operands. We then realized there was no reason to restrict constructors to specifying encoding procedures, so we made it possible to apply constructors both in pattern definitions and in matching statements, yielding the language described in this paper.

Patterns are a simple yet powerful way to describe the binary formats of instructions. Field constraints, conjunction, and concatenation are all found in architecture manuals, and together they can describe any instruction on any of the three machines we have studied. They're not limited to traditional instruction sets in which opcode and operand are clearly separated; all three machines use instruction formats in which opcode bits are scattered throughout the instruction. Disjunction does not make it possible to specify new instructions, but it improves specifications by making it possible to combine descriptions of related instructions. By removing the need to specify each instruction individually, disjunction eliminates a potential source of error.

If patterns provide a good high-level description of binary encodings, constructor specifications raise the level of abstraction to that of assembly language. Equations, though seldom used, are needed to describe instructions like relative branches, whose assembly-level operands differ from their machine-level fields. Equations can

also express constraints, which are part of the definitions of some architectures, like the Intel 486.

We maximize the power of the toolkit's specification language by minimizing restrictions on the way patterns, constructors, and equations can be combined. For example, patterns and constructors can be used in each other's definitions, which makes it possible to factor complex architectures like the 486. Equations in constructor specifications are used for both encoding and decoding, and equations can also be used in matching statements. Because the parts of the language work together, it is hard to see how the language could be simplified without destroying it. The only obvious candidate for removal is the conditional-assembly construct, but it is a natural extension of using equations to specify constructors, requiring only a little extra syntax. The simplicity of the specifications and the checking done by the toolkit combine to give users confidence in the correctness of the generated code.

Acknowledgements

This work has been funded by a Fannie and John Hertz Fellowship, an AT&T PhD Fellowship, an IBM Graduate Research Fellowship, and by Bellcore. More colleagues than we have room to mention were kind enough to review preliminary versions of this paper. We are especially indebted to David Keppel and Karin Petersen for their thorough readings and suggestions.

Author information

After a fleeting career as a physicist, Norman Ramsey (norman@bellcore.com) returned to computing in 1986. He likes to build programming tools that people actually use. He received the PhD degree from Princeton University in 1993. Mary Fernandez (mff@cs.princeton.edu) is a PhD candidate at Princeton University and is an avid user of Norman's programming tools.

The New Jersey Machine-Code Toolkit is available by anonymous ftp from ftp.cs.princeton.edu in directory pub/toolkit. Its home page on the World-Wide Web is http://www.cs.princeton.edu/software/toolkit.

This paper was prepared using the noweb tools for literate programming (Ramsey 1994b). The SPARC examples have been checked for consistency with Appendix A. All of the examples have been extracted from the paper and run through the toolkit, and they all work with version 0.1a.

References

Ball, Thomas and James R. Larus. 1992 (January). Optimially profiling and tracing programs. In Conference Record of the ACM Symposium on Principles of Programming Languages, pages 59–70, Albuquerque, NM.

- Bell, C. G. and A. Newell. 1971. Computer Structures: Readings and Examples. New York: McGraw-Hill.
- Bell, Timothy C., John G. Cleary, and Ian H. Witten. 1990. Text Compression. Englewood Cliffs, NJ: Prentice Hall.
- Duffy, Diane E., Allen A. McIntosh, Mark Rosenstein, and Walter Willinger. 1994 (April). Statistical analysis of CCSN/SS7 traffic data from working CCS subnetworks. *IEEE Journal on Selected Ar*eas in Communications, 12(3):544-551.
- Elsner, Dean, Jay Fenlason, et al. 1993 (March). *Using* as: the GNU Assembler. Free Software Foundation.
- Ferguson, David E. 1966. The evolution of the meta-assembly program. Communications of the ACM, 9(3):190-193.
- Fernandez, Mary. 1994 (November). Simple and effective link-time optimization of Modula-3 programs. Technical Report CS-TR-474-94, Department of Computer Science, Princeton University.
- Fraser, Christopher W. and David R. Hanson. 1991 (October). A retargetable compiler for ANSI C. SIGPLAN Notices, 26(10):29-43.
- Fraser, Christopher W., Robert R. Henry, and Todd A. Proebsting. 1992 (April). BURG—fast optimal instruction selection and tree parsing. SIGPLAN Notices, 27(4):68-76.
- Griswold, Ralph E. and Madge T. Griswold. 1990. *The Icon Programming Language*. Second edition. Englewood Cliffs, NJ: Prentice Hall.
- Hastings, Reed and Bob Joyce. 1992 (January). Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, pages 125–136.
- Intel Corporation. 1990. i486 Microprocessor Programmer's Reference Manual.
- International Organization for Standardization. 1987.

 Information Processing Open Systems Interconnection Specification of Abstract Syntax
 Notation One (ASN.1). ISO 8824 (CCITT
 X.208).
- Nelson, Greg, editor. 1991. Systems Programming with Modula-3. Englewood Cliffs, NJ: Prentice Hall.
- Ramsey, Norman and David R. Hanson. 1992 (July). A retargetable debugger. ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, in SIGPLAN Notices, 27(7):22– 31.
- Ramsey, Norman. 1992 (December). A Retargetable Debugger. PhD thesis, Princeton University, Department of Computer Science. Also Technical Report CS-TR-403-92.
- ——. 1994a (January). Correctness of trap-based breakpoint implementations. In Proceedings of the 21st ACM Symposium on the Principles of Programming Languages, pages 15-24, Portland, Oregon.

- ——. 1994b (September). Literate programming simplified. *IEEE Software*, 11(5):97–105.
- SPARC International. 1992. The SPARC Architecture Manual, Version 8. Englewood Cliffs, NJ: Prentice Hall.
- Svrivastava, Amitabh and David W. Wall. 1993. A practical system for intermodule code optimization. *Journal of Programming Languages*, 1:1–18. Also available as WRL Research Report 92/6, December 1992.
- Wick, John Dryer. 1975 (December). Automatic Generation of Assemblers. PhD thesis, Yale University.

A Partial SPARC spec

This partial specification of the SPARC includes all the examples in the paper. It omits many floating-point instructions, several flavors of load, store, read and write instructions, and many synthetic instructions. The reader is encouraged to compare the specification to the equivalent information provided in the SPARC architecture manual (SPARC 1992). Page references are provided for cross reference.

This fields declaration defines the fields used in all SPARC instructions:

```
fields of itoken (32)
op 30:31 rd 25:29 op3 19:24 rs1 14:18
i 13:13 simm13 0:12 opf 5:13 rs2 0:4
op2 22:24 imm22 0:21 a 29:29 cond 25:28
disp22 0:21 asi 5:12 disp30 0:29
```

The following patterns represent Tables F-1 and F-2 on p 227.

```
patterns
[TABLE_F2 call TABLE_F3 TABLE_F4]
  is op = {0 to 3}
[unimp _ Bicc _ sethi _ fbfcc cbccc ]
  is TABLE_F2 & op2 = {0 to 7}
```

The following patterns represent Table F-3 on p 228.

patt	erns			
[add	addcc	taddcc	WYXXX
	and	andcc	tsubcc	wrpsr
	or	orcc	taddcctv	wrwim
	xor	xorcc	tsubcctv	wrtbr
	sub	subcc	mulscc	fpop1
	andn	andncc	sll	fpop2
	orn	orncc	srl	cpop1
	xnor	xnorcc	sra	cpop2
	addx	addxcc	rdxxx	jmpl
	_	() ()	rdpsr	rett
	umul	umulcc	rdwim	ticc
	smul	smulcc	rdtbr	flush
	subx	subxcc		save
	_	2-2	_	restore
	udiv	udivcc	-	_
	sdiv	sdivcc	_	_]
is				

TABLE_F3 & op3 = { 0 to 63 columns 4 }

The following patterns represent Table F-4 on p 229.

```
[ ld
          lda
                   1df
                          1dc
  ldub
          lduba
                   ldfsr ldcsr
  lduh
          lduha
                   lddf lddc
  144
          ldda
                   stf
                          stc
  st
          sta
  stb
          stba
                   stfsr stcsr
  sth
          stha
                   stdfq stdcq
          stda
                   stdf stdc
  std
  ldsb
         ldsba
  ldsh
          ldsha
  ldstub ldstuba
                             1
  swap
          swapa
  TABLE_F4 & op3 = \{0 \text{ to } 63 \text{ columns } 4\}
```

The unimp pattern is used as a place holder in the instruction stream for instructions that refer to unknown relocatable addresses.

placeholder for itoken is unimp & imm22 = 0xbad

Address operands, defined on p 84, have four possible formats.

constructors

```
dispA rs1 + simm13! : Address
is i = 1 & rs1 & simm13

absoluteA simm13! : Address
is i = 1 & rs1 = 0 & simm13

indexA rs1 + rs2 : Address
is i = 0 & rs1 & rs2

indirectA rs1 : Address
is i = 0 & rs2 = 0 & rs1
```

Register or immediate operands, defined on p 84, have two possible formats.

constructors

```
rmode rs2 : reg_or_imm is i = 0 & rs2
imode simm13! : reg_or_imm is i = 1 & simm13
```

The following example specifies the assembly syntax and binary encoding for all load-integer instructions defined on p 90. It shows that Address operands are delimited by brackets in the assembly language.

The following patterns group the logical, shift, and arithmetic opcodes.

| udiv | sdiv | udivcc | sdivcc

alu is arith | logical | shift

The assembly syntax and binary encoding for all alu instructions is the same.

| save | restore | taddcctv | tsubcctv

constructors

```
alu rs1, reg_or_imm, rd
```

The following pattern represents the first column of Table F-7 on p 231.

patterns

```
branch is any of

[ bn be ble bl bleu bcs bneg bvs

ba bne bg bge bgu bgeu bpos bvc ],

which is Bicc & cond = {0 to 15}
```

where

```
p is any of [a \ b \ \dots z], which is generating expression is syntactic sugar for [a \ b \ \dots z] is generating expression p is a \ | \ b \ | \ \dots \ | \ z
```

The synthetic instructions bset and dec are defined on p 86. They are assembled using the machine instructions or and sub.

constructors

```
bset reg_or_imm, rd is or(rd, reg_or_imm, rd)
dec val, rd is sub(rd, imode(val), rd)
```

The assembly syntax for branch instructions is defined on pp. 119-120.

constructors

```
branch reloc { reloc = $pc + 4 * disp22! }
  is branch & disp22
```

The conditionally assembled instruction set is defined on p 84. This definition attempts to assemble set into a single instruction when possible.

```
constructors
```

ATOM A Flexible Interface for Building High Performance Program Analysis Tools

Alan Eustace and Amitabh Srivastava

Digital Equipment Corporation Western Research Laboratory

Abstract

Code instrumentation is a powerful mechanism for understanding program behavior. Unfortunately, code instrumentation is extremely difficult, and therefore has been mostly relegated to building special purpose tools for use on standard industry benchmark suites.

ATOM (Analysis Tools with OM) provides a very flexible and efficient code instrumentation interface that allows powerful, high performance program analysis tools to be built with very little effort. This paper illustrates this flexibility by building five complete tools that span the interests of application programmers, computer architects, and compiler writers.

This flexibility does not come at the expense of performance. Because ATOM uses procedure calls as the interface between the application and the analysis routines, the performance of each tool is similar to or greatly exceeds the best known hand-crafted implementations.

1. Introduction

Program analysis tools are extremely important for understanding program behavior. Computer architects need such tools to evaluate how well programs will perform on new architectures. Software writers need tools to analyze their programs and identify critical pieces of code. Compiler writers often use such tools to determine how well their instruction scheduling or branch prediction algorithm is performing or to provide input for profile-driven optimizations.

Over the past decade three classes of tools for different machines and applications have been developed. The first class consists of basic block counting tools like Pixie[13], Epoxie[22] and QPT[11]. The second class consists of data and instruction address tracing

tools. Pixie and QPT can also generate address traces. They communicate these traces to analysis routines through inter-process communication. Tracing on the WRL Titan[3] communicated with analysis routines using shared memory, but this required operating system modifications. MPTRACE [6] is similar to Pixie but it collects traces for multiprocessors by instrumenting assembly code. ATUM [1] generates address traces by modifying microcode and saves a compressed trace in a file that is analyzed offline. The third class of tools consists of simulators. Tango Lite[7] supports multiprocessor simulation by instrumenting assembly language code. PROTEUS[4] also supports multiprocessor simulation but instrumentation is done by the compiler. g88[2] simulates Motorola 88000 using threaded interpreter techniques. Shade[5] uses instruction level simulation to selectively generate traces. This technique offers considerable flexibility at the expense of much lower performance.

The important features that distinguish ATOM[18, 15, 16] from previous systems are listed below.

- ATOM is a tool-building system. A diverse set of tools ranging from basic block counting to cache modeling can be easily built.
- ATOM provides the common infrastructure in all code-instrumenting tools, which is the cumbersome part. The user simply specifies the tool details.
- ATOM allows selective instrumentation. The user specifies the points in the application to be instrumented, the procedure calls to be made, and the arguments to be passed.
- The communication of data is through procedure calls. Information is *directly* passed from the application to the specified analysis routine with a

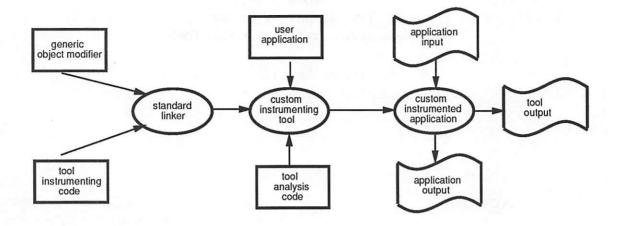


Figure 1: The ATOM Process

procedure call instead of through interprocess communication, files on disk, or a shared buffer with central dispatch mechanism.

- ATOM tool overhead is proportional to the complexity of the underlying analysis. Many interesting tools can be built that have little or no impact on application performance.
- Even though the analysis routines run in the same address space as the application, precise information about the application is presented to analysis routines at all times.
- ATOM is independent of compiler and language systems.

To illustrate the power and flexibility of this approach, this paper fully implements a variety of custom program analysis tools, including input/output, instruction profiling, cache simulation, dynamic memory allocation, procedure inlining profile driven optimizations, and evaluating the quality of compiled code. None of these tools takes more than 60 lines of code to implement. These tools form the basis of many of the tools that are distributed as part of the standard ATOM distribution. Digital provides ATOM as an Advanced Development Kit in OSF/1, Version 3.

To illustrate the performance of these tools, each was applied to the SPEC92 tool suite. The instrumented

application times are compared to the uninstrumented applications using wall clock times.

2. Implementation of ATOM

ATOM is built using OM[19], a link-time code modification system. OM takes as input a fully linked executable that has been annotated with relocation records. This file is used to build a symbolic intermediate representation which contains no hard coded addresses. This framework provides an ideal platform for inserting, rearranging, or deleting instructions. Previous papers [20, 21] have explored the usefulness of this approach at performing a wide range of program optimizations. ATOM uses this same intermediate representation to implement program instrumentation. A complete description of the ATOM implementation was presented at PLDI[18].

Figure 1 describes the implementation. First, the OM generic object modification library is linked with a tool specific *instrumentation* file to produce a custom instrumenting tool. This program reads in the user application, and modifies it by adding calls to tool specific analysis procedures. ATOM completes the process by linking the instrumented application with the tool specific *analysis* file. The output of ATOM is a custom instrumented application executable that is run in exactly the same manner as the original application.

```
#include <stdio.h>
  #include <instrument.h>
3
   Instrument() {
4
     Proc *p;
     AddCallProto("RecordRead(REGV)");
5
     AddCallProto("PrintResults()");
6
     p = GetNamedProc("read");
7
     if (p != NULL) {
8
       AddCallProc(p,ProcAfter,"RecordRead", RET_RES_1);
9
       AddCallProgram(ProgramAfter,"PrintResults");
10
11
     }
12 }
```

Analysis File

```
1 #include <stdio.h>
2 long bytes = 0;
3 void RecordRead(long size) {
4 bytes = bytes + size;
5 }
6 void PrintResults() {
7 FILE *file = fopen("read.out","w");
8 fprintf(file, "%ld\n", bytes);
9 fclose(file);
10 }
```

Figure 2: Read Tool Implementation

3. A Simple Example

From a user perspective, applying an ATOM tool to an application is done by executing the following command.

atom appl.rr read.inst.c read.anal.c -o appl.read

The first argument is the application program, which has been specially linked to include relocation records. The second and third arguments are the *instrumentation* and *analysis* files. In this example, we instrument the application to count and write to a file the total number of bytes read each time the instrumented application is executed.

The instrumentation file is shown of the left side of Figure 2. Line 2 includes the *instrument.h* file which defines the ATOM primitives for manipulating application programs. Line 3 defines the Instrument procedure, which is linked with the OM object code modification library to produce a custom instrumenting tool. All analysis procedures that are called from the application program are declared and placed by the Instrument procedure.

Line 5 makes use of the AddCallProto ATOM primitive to declare the name and arguments to the RecordRead analysis procedure. This procedure takes a single argument of type REGV. Arguments of type REGV are used to pass the contents of a specific processor register. Line 6 declares the PrintResult analysis procedure, which does not take any arguments.

Line 7 calls the GetNamedProc primitive to return a pointer to the read procedure. Line 8 checks to see if this value is NULL, indicating that the procedure read is not defined in this application. All communication between the application program and the analysis pro-

cedures is done through procedure calls.

Line 9 uses the ATOM primitive AddCallProc to add a call to the read procedure. The first argument, p is a pointer to the read procedure. The second argument, ProcAfter, specifies that the call is to be inserted after the read procedure is executed. The third argument indicates that the call is to the RecordRead analysis procedure. The remainder of the arguments are used to determine what values ATOM passes to RecordRead. In this case, the final argument passes the contents of the register RET_RES_1 to the analysis procedure. In the Alpha AXP calling convention, this register contains the contents of the value returned by the read procedure. The RecordRead procedure is shown on the right side of Figure 2. This procedure simply adds this size to a total.

Line 10 calls the AddCallProgram primitive, which adds a call to the PrintResults procedure after the application finishes executing. The corresponding analysis procedure opens a file, prints out the result, and closes the file. The definitions for both these procedures are shown on the right side of Figure 2. A sample output file is shown in the Appendix.

Notice that it is important that the analysis procedure does not call the instrumented version of the read procedure, since reads that occur inside the analysis procedure must not increment the application totals. To guarantee this, library procedures that would normally be shared between the application and the analysis procedures are linked into the instrumented application twice. Only the version that is linked into the application is instrumented. This guarantees that calls made to read by the analysis procedures do not influence the statistics gathered by the read tool.

Although this tool is relatively simple, it is straightforward to extend the read tool into a general input/output tool. The first extension is to add calls to analysis procedures before and after for open system calls. This allows the analysis procedures to record the name of the file opened and the file descriptor returned. By instrumenting both read and write procedures and passing the first (file descriptor) and third arguments (size in bytes), the read and write totals can be accumulated for each open file. The final extension is to use the Alpha AXP cycle counter to maintain fine grain times of how long each operation takes. This allows the tool to determine the rate of read and write operations. This extended tool is called *io* and is distributed with ATOM as part of the standard tool set.

4. ATOM Primitives

ATOM tools traverse an application, find interesting places to add calls to analysis procedures, and pass arguments that correspond to data or events in the application. To provide these functions, ATOM provides three types of primitives: *navigation*, *information*, and *instrumentation*.

Navigation primitives traverse the application. The simple example presented above used the GetNamedProc primitive to find a specific procedure. Other navigational primitives traverse procedures, basic blocks within procedures, and instructions within basic blocks. A *basic block* is a set of sequential assembly language instructions that are not interrupted by branches or jumps.

Information primitives provide static information about instructions, basic blocks, procedures, or the program. For example, given an instruction, ATOM primitives can return the program counter, the opcode, the instruction class, address displacements, the source line number, a mask of the registers used or set by the instruction, etc. Given a basic block, primitives are provided to find the number of instructions in the basic block and the starting program counter of the block. Given a procedure, primitives are provided to find the file name, stack frame size, register save and restore masks, etc. General program information includes the sizes of text and data sections, along with general statistics on the number of procedures, basic blocks and procedures in the application.

Instrumentation primitives allow calls to analysis procedures to be inserted into the application before or after

instructions, basic blocks, procedures. The arguments to these procedures can include any value computed by the instrumentation routine or provided by ATOM primitives. The arguments of these procedures can be constants, processor registers, effective addresses, branch condition values, arguments to application procedures, file names, line numbers, or character strings.

Although not shown in these examples, ATOM also allows command line arguments to be passed to instrumentation routines. Parameters can also be passed to analysis procedures through *seteny* variables.

5. Instruction Profiling

In this section we implement an instruction profiler based on counting the number of instructions executed in each procedure. Although it is possible to implement this tool by placing a call to an analysis procedure before every instruction in the application, ATOM's selective instrumentation can significantly reduce this overhead by instrumenting only basic blocks. For example, if a set of 10 sequential executed instructions are inside of a loop, we can keep track of the total number of instructions executed by adding 10 each time we enter the loop body.

Figure 3 defines the instrumentation and analysis files for the profile tool. As in the previous section, lines 6 through 9 of the instrumentation file declares the interface to the OpenFile, ProcedureCount, ProcedurePrint, and CloseFile analysis procedures.

In line 10, the AddCallProgram primitive is used to add a call to OpenFile before the application begins execution. The GetProgramInfo ATOM primitive, when passed the ProgramNumberProcs argument, returns the number of procedures in the application. The corresponding analysis procedure uses this argument to allocate sufficient memory to accumulate a count for each procedure in the application.

Lines 12 through 21 navigate each procedure in the application. Within each procedure, lines 14 through 18 process each basic blocks. Line 16 calls the AddCallBlock ATOM primitive to add a call to the ProcedureCount analysis procedure. The two arguments passed are a procedure index n, and the number of instructions in the basic block. This value is returned by the GetBlockInfo primitive. The corresponding analysis procedure uses these arguments to increment the number of instructions executed by this procedure.

```
#include <stdio.h>
2
   #include <instrument.h>
3
  Instrument() {
     Proc *p; Block *b;
4
5
     int n = 0;
6
     AddCallProto("OpenFile(int)");
7
     AddCallProto("ProcedureCount(int,int)");
8
     AddCallProto("ProcedurePrint(int,char *)");
9
     AddCallProto("CloseFile()");
10
     AddCallProgram(ProgramBefore, "OpenFile",
           GetProgramInfo(ProgramNumberProcs));
11
     for (p = GetFirstProc(); p != NULL;
12
           p = GetNextProc(p)) {
13
       for (b = GetFirstBlock(p); b != NULL;
14
             b = GetNextBlock(b)) {
15
         AddCallBlock(b,BlockBefore, "ProcedureCount",
16
             n,GetBlockInfo(b,BlockNumberInsts));
17
18
       AddCallProgram(ProgramAfter, "ProcedurePrint",
19
              n++, ProcName(p));
20
21
     AddCallProgram(ProgramAfter, "CloseFile");
22
23 }
```

Analysis File

```
#include <stdio.h>
   long instrTotal;
   long *instrPerProc;
   FILE *file;
   void OpenFile(int n) {
5
     instrPerProc = (long *) malloc(sizeof(long) * n);
6
7
     file = fopen("prof.out", "w");
     fprintf(file, "%30s %15s %10s\n", "Procedure",
8
            "Instructions", "Percentage");
9
10 }
11 void ProcedureCount(int n, int instructions) {
     instrTotal += instructions;
     instrPerProc[n] += instructions;
13
14 }
15 void ProcedurePrint(int n, char *name) {
     if (instrPerProc[n] > 0)
        fprintf(file, "%30s %15ld %9.3f\n", name,
17
       instrPerProc[n], 100.0 * instrPerProc[n] / instrTotal);
18
19 }
20 void CloseFile() {
     fprintf(file, "\n%30s %15ld\n", "Total", instrTotal);
     fclose(file);
23 }
```

Figure 3: Profiling Tool Implementation

For each procedure in the application, line 19 adds a call to the ProcedurePrint analysis procedure. ProcedurePrint is passed the unique procedure index and the name of the procedure. This name is returned by the ProcName ATOM primitive. The corresponding analysis file uses these two parameters to determine if the procedure was executed, and if so, prints the procedure name, number of instructions, and percentage of instructions executed in this procedure to a file. Notice that the effect of line 19 is to add hundreds of calls to analysis procedures to the end of the program, each with a different index and character string.

Line 22 adds a call to the CloseFile analysis procedure after the application completes executing. A sample output file is included in the Appendix.

Although this is a very simple profiling tool, many more interesting tools can be built using the same principles. Russell Kao built an ATOM based version of the popular tool *Gprof*. This tool adds procedure calls at the start of each procedure to push the name of the procedure on a procedure call stack. This stack is popped by adding a similar analysis procedure call to the procedure exit. Gprof reports the percentage of time spent in a procedure and the procedures descendants. The

instrumentation procedure was also expanded to use the Alpha AXP dual issue rules to compute cycles rather than instructions executed.

Many other profile based tools have also been developed. One such tool records the value of the Alpha AXP cycle counter at the start of the procedure and at the end of the procedure and computes the wall clock time spent in each procedure.

6. Cache Simulator

Processor cycle times are getting faster at a much greater rate than main memory access times. This disparity has led computer architects to place a subset of main memory into one or more levels of fast, expensive cache memory [9]. The effectiveness of this technique is application dependent. Applications that reference the same address multiple times or that use nearby data items benefit most from the data cache.

Although it is clear that cache memory plays an increasingly important role in application performance, measuring cache performance has been relegated to a few industrial and university research reports. Almost all of these studies have focused primarily on the per-

```
1
   #include <stdio.h>
2
   #include <instrument.h>
3
   Instrument() {
4
     Proc *p; Block *b; Inst *i;
5
     AddCallProto("Reference(VALUE)");
     AddCallProto("PrintResults()");
7
     for (p = GetFirstProc(); p != NULL;
            p = GetNextProc(p)) {
9
       for (b = GetFirstBlock(p); b != NULL;
10
              b = GetNextBlock(b)) {
11
          for (i = GetFirstInst(b); i != NULL;
12
                i = GetNextInst(i)) {
13
            if (IsInstType(i,InstTypeLoad) ||
14
                IsInstType(i,InstTypeStore))
15
              AddCallInst(i,InstBefore,
16
                  "Reference", EffAddrValue);
17
          }
18
       }
19
20
     AddCallProgram(ProgramAfter, "PrintResults");
21 }
```

Analysis File

```
#include <stdio.h>
1
2
   #define CACHE_SIZE 65536
3
   #define BLOCK_SHIFT 5
   long cache[CACHE_SIZE >> BLOCK_SHIFT];
4
5
   long references, misses;
6
   void Reference(long address) {
7
     int index =
9
         address & (CACHE_SIZE-1)) >> BLOCK_SHIFT;
10
     long tag = address >> BLOCK_SHIFT:
     if (cache[index] != tag) {
11
12
       misses++:
13
       cache[index] = tag;
     }
14
15
     references++;
16 }
17 void PrintResults() {
18
     FILE *file = fopen("cache.out", "w");
19
     fprintf(file, "%ld %ld %f\n",
20
       references, misses, 100.0 * misses / references);
21
     fclose(file);
22 }
```

Figure 4: Cache Tool Implementation

formance of the SPEC92 benchmark suite.

This section presents a simple tool that simulates the execution of the application running in a 64K-byte direct mapped data cache with 32-byte blocks. The tool computes the total number of data cache references, the number of misses, and the *miss rate*. The miss rate is the number of misses divided by the number of references.

The strategy used in this tool is to instrument all load and store instructions with a call to an analysis procedure called Reference which is passed the effective address. This effective address is used to simulate the application running in the cache. The cache tool implementation is shown in Figure 4.

Line 5 of the instrumentation file declares the Reference analysis procedure. The type VALUE indicates that the argument does not live in a processor register, but must be computed by ATOM prior to passing the value to the analysis procedure. Lines 11 through 17 examine each instruction. Lines 13 and 14 determine if the instruction is a load or a store. If so, the AddCallInst ATOM primitive adds a call to instruction i. The InstBefore argument adds the call before the instruction. The name of the analysis procedure to call is Reference, and the argument passed is the EffAddrValue, which ATOM computes by adding the contents of the base register plus the sign

extended displacement. Line 20 completes the tool by adding a call to the PrintResults procedure after the application completes execution.

The analysis procedure is shown on the right side of Figure 4. This is a simple implementation of a direct mapped cache. Lines 4 defines the cache data structure, which is used to hold a cache tag for each 32 byte block in the cache. Line 5 defines the reference and miss counters. Lines 7 through 9 compute the cache tag and index, and line 11 probes the cache. If the tags do not match, a miss is recorded in line 12, and the tag is updated in line 13. In either case, the number of references is incremented. A sample output file is shown in the Appendix.

To guarantee that the results properly reflect the reference pattern of the uninstrumented program, ATOM guarantees that all data items referenced in the original program are placed in exactly the same locations when the program is instrumented. To guarantee this accuracy for instruction cache simulations, ATOM converts all references to the program counter to those of the uninstrumented program before passing the contents to the analysis procedures.

Although the number of hits and misses is useful to computer architects, this information has rarely been presented in a form that is useful to application pro-

```
1 #include <stdio.h>
  #include <instrument.h>
2
  void Instrument() {
3
     Proc *procMalloc =
             GetNamedProc("malloc");
5
     Proc *procFree = GetNamedProc("free");
6
     AddCallProto("PrintResults()");
7
8
     if (procMalloc)
       ReplaceProcedure(procMalloc, "my_malloc");
9
10
     if (procFree)
11
       ReplaceProcedure(procFree, "my_free");
     AddCallProgram(ProgramAfter, "PrintResults");
12
13 }
```

Analysis File

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 long totalMalloc, totalFree = 0;
4 char *my_malloc(size_t size) {
     size_t *mptr = (long *) malloc(size+sizeof(long));
     totalMalloc += size;
6
     mptr[0] = size;
8
     return ((void *) &mptr[1]);
9 }
10 my_free(void *ptr) {
     size_t *mptr = ptr;
     size\_t size = mptr[-1];
13
     totalFree += size;
14
     free(&mptr[-1]);
15 }
16 void PrintResults() {
17 FILE *file = fopen("dyn.out", "w");
18 fprintf(file, "%ld %ld\n", totalMalloc, totalFree);
19 fclose(file);
20 }
```

Figure 5: Dynamic Memory Tool Implementation

grammers. By combining the instruction profile tool shown in the previous section with the cache modeling tool shown above, ATOM can create a hybrid tool that shows cache misses in a profile like format. This tool is called *memsys* and it is included with the standard ATOM distribution.

7. Dynamically Allocated Memory

Many programs make extensive use of dynamically allocated memory. Such memory is typically allocated using the malloc system call, and deallocated using the free system call. These procedures are called thousands of times by application programs, allocating, deallocating, and reallocating the same piece of memory many times. This section presents a tool that computes the total number of bytes allocated and freed over the course of the application's execution.

The implementation is shown in Figure 5. Lines 4 through 6 of the instrumentation file are used to search for procedures with the names malloc and free. If these procedures are present in the application, these library functions are replaced in lines 6 and 7 by the procedures my_malloc and my_free. The ReplaceProcedure semantics require the type and arguments of the new procedure to be identical to

the original procedure calls.

The analysis procedures prepend the size of allocated objects to each dynamically allocated element. Line 5 calls the standard version of malloc, but requests additional memory to prepend the object size. Line 6 adds this size to the total amount of allocated memory. Line 7 saves this size in the first location in the dynamically allocated memory. The pointer to the start of the requested memory is returned in line 8. Each call to free was replaced in the application by a call to my_free. In line 12, this procedure uses a negative index to access the size of the object, which it adds to the total amount deallocated by the application. Line 14 calls the standard free procedure to deallocate the memory. A sample output file is shown in the appendix.

The ability to replace procedures and monitor data references is fundamental to an emerging set of tools that monitor allocations, deallocations and references to memory[8]. Jeremy Dion and Louis Monier[14] recently completed an ambitious ATOM based tool called Third Degree, that finds and reports many kinds of reads of uninitialized memory, reads and writes to unallocated memory, array bound errors, and freeing the same object more than once. The technique used is to replace all calls to allocate and free library procedures with versions that keep track of the ranges of valid heap lo-

Instrumentation File **Analysis File** 1 #include <stdio.h> 2 #include <instrument.h> #include <stdio.h> 3 Instrument() { struct Work { Proc *p; Block *b; Inst *i; 3 long count; 5 int n = 1; 4 long wasted; 6 AddCallProto("OpenFile(int)"); 5 } *work; AddCallProto("SaveLoad(REGV)"); 7 6 FILE *file; AddCallProto("CheckLoad(int,long)"); void OpenFile(int n) { 8 7 9 AddCallProto("Print(int,long)"); 8 work = (struct Work *) 10 AddCallProto("CloseFile()"); 8 malloc(sizeof(struct Work) * n); for (p = GetFirstProc(); p != NULL; 9 11 file = fopen("work.out", "w"); 12 p = GetNextProc(p)) { 10 fprintf(file, "%11s %11s %11s\n", 13 for (b = GetFirstBlock(p); b != NULL; 10 "PC", "Count", "Wasted"); 14 b = GetNextBlock(b)) { 11 } 15 for (i = GetFirstInst(b); i != NULL; 12 void CloseFile() { 16 15 fclose(file); i = GetNextInst(i) { 17 if (IsInstType(i,InstTypeLoad)) { 16 } 18 AddCallInst(i,InstBefore, "SaveLoad", 17 long value; 19 GetInstRegEnum(inst,InstRA)); 18 void SaveLoad(long val) { 20 AddCallInst(i,InstAfter, "CheckLoad", value = val; 19 21 n, GetInstRegEnum(inst,InstRA)); 20 } 22 AddCallProgram(ProgramAfter, "Print", 21 void CheckStore(int n,long val) { 23 n++, InstPC(i)); work[n].count++; 24 if (value == val) work[n].wasted++; 25 24 } 26 25 void Print(int n, long pc) { 27 if (work[n].wasted != 0)28 AddCallProgram(ProgramBefore, "OpenFile",n); fprintf(file, " $0x\%91x\%111d\%111d\n$ ", 29 29 29 pc, work[n].count, work[n].wasted); AddCallProgram(ProgramAfter, "CloseFile"); 30 } 30 }

Figure 6: Compiler Auditing Implementation

cations. Symbolic interpretation in the instrumentation procedures is used to significantly reduce the number of memory references that must be instrumented. The result is a very effective and efficient tool for testing the validity of memory operations. This tool is also included in the standard ATOM distribution.

8. Compiler Auditing

Modern compilers implement a long list of optimizations: loop unrolling, reductions in strength, software pipelining, global register allocation, instruction rearrangement. Unfortunately, these techniques are complicated and interact in non-trivial ways. The resulting code often misses simple optimizations. Tools that evaluate the quality of the compiled code and isolate potential performance problems are called *compiler auditors*[12].

This section presents a simple compiler auditing tool that adds a procedure call before each load instruction to save the contents of the destination register. Another procedure call is added after each load instruction that checks to see if the destination register was modified by the instruction. If not, the instruction loaded a value that was already in the register. These loads are termed redundant.

The implementation is shown in Figure 6. This tool is similar to previous tools, with the exception of lines 17 through 24. Line 17 checks if the instruction is a load operation. If so, line 18 adds a call to the SaveLoad procedure before the instruction and passes the contents of the destination register, as returned by the GetInstRegEnum ATOM primitive. Line 20 adds a matching call to CheckLoad after the load instruction. The arguments are a unique index of the load instruction, and the new contents of the destination register. CheckLoad compares this value to the value saved by

the SaveLoad analysis procedure and increments the appropriate counters. The output file contains a count of the number or redundant times each load is executed along with the number of times the load was redundant. A sample output file is shown in the appendix.

This tool is very effective at detecting loop invariant instructions and unnecessary spilling and restoring of registers. In one very early version of the compiler, this tool found 8 identical sequential load instructions from the same memory location to the same destination register!

Redundant loads are often caused by redundant data, and therefore are not be indicative of compiler performance bugs. This is the case in the SPEC92 benchmark *hydro2d* where an amazing 42 percent of the loads are redundant. This is caused by large sparsely populated arrays.

9. Performance

The performance of ATOM tools is a function of the number of analysis procedure calls that are executed and the amount of work done by each call. Figure 7 shows the performance of each tool over the SPEC92 benchmark suite. Each entry reflects the wall clock time of the instrumented program divided by the wall clock time of the uninstrumented program.

The Dynamic Memory and Read tools have a minimal affect on application performance, since both have relatively few instrumentation points. Contrast this with the compiler auditing tool, which adds two calls to analysis procedures for each load instruction. Also notice that there is considerable variation between benchmarks for a single tool. For example, the profile tool slows down application by as little as 1.472 for swm256 and as much as 8.919 for espresso. Both instrument at basic blocks, but since the basic block size of espresso is much smaller, the instrumented application spends a larger percentage of time in the analysis procedures. These slowdowns have been acceptable for the vast majority of applications. However, in real time applications, ATOM's selective instrumentation interface can be used to minimize the instrumentation effects on program behavior.

When comparing these times to other tools reported in the literature, it is important to include the time necessary to gather the data *and* to analyze the results. For example, many cache instrumentation tools studies report competitive times for gathering trace data into in-

	DynMem	Read	Prof	Cache	Audit
alvinn	0.987	0.989	3.91	8.71	11.93
compress	1.007	0.980	4.14	9.37	7.52
doduc	0.990	1.008	2.91	8.30	12.05
ear	1.011	1.005	5.75	6.61	9.40
eqntott	0.995	0.997	4.18	8.12	10.84
espresso	1.050	1.006	8.92	10.66	14.49
fpppp	0.994	0.994	1.76	12.97	21.83
gcc1	1.016	1.006	5.65	8.60	10.54
hydro2d	0.987	0.991	2.40	7.42	10.51
li	1.021	1.054	6.20	11.06	12.48
mdljdp2	0.996	1.000	2.93	4.32	5.20
mdljsp2	1.020	1.027	3.18	6.13	9.37
nasa7	0.997	1.002	1.69	8.19	11.38
ora	0.931	0.931	4.22	7.88	11.65
sc	1.033	1.030	6.01	6.96	8.50
spice	1.012	1.018	3.93	7.28	9.54
su2cor	0.985	0.992	2.46	7.73	9.64
swm256	0.987	0.990	1.47	9.54	13.96
tomcatv	0.998	0.990	1.82	6.40	13.55
wave5	0.995	0.993	3.15	9.28	10.69
Average	1.000	1.004	3.61	8.27	11.23

Figure 7: Performance of Atom Tools

memory buffers, but do not include the times to empty the buffer, simulate the cache, and report the results.

There are many ways to substantially increase the performance of ATOM based tools. One approach is to reduce or eliminate the analysis procedure call overhead either through inlining or other compiler optimizations. Another approach is to make use of the flexibility of the instrumentation interface to reduce the frequency of analysis procedure calls. For example, the profile tool instrumentation routine can be easily rewritten to eliminate adding calls to analysis procedures for those blocks where data flow analysis determines that the count is identical to another block that has already been instrumented. Another example is instruction translation buffer simulation. Here, ATOM based tools need only instrument branches or sequential execution that crosses page boundaries. Since these are relatively infrequent, these tools are very efficient. Another example are tools that simulate branch prediction algorithms. Rather than infer branch behavior by sifting through instruction address traces, ATOM tools instrument only conditional branches.

10. Conclusions

ATOM is a unique tool for understanding program performance. The flexible interface allows a diverse set of tools to be built with minimal effort. Without the support ATOM provides, these tools would be extremely difficult to build. The performance of these tools compares favorably with hand-crafted implementations, since instrumentation is inserted only when necessary to gather statistics. Communication of data to the analysis procedures is accomplished through procedure calls, rather than relying on expensive interprocess communication. The analysis routines are always presented with information about the application program as if it was executing uninstrumented.

ATOM has been applied to many commercial applications with text sizes of up to 100MB. Hundreds of tools have been written by both industrial and university users to evaluate the performance of caches, garbage collection algorithms, branch prediction, compiler optimizations, input/output, system calls, novel CPU architectures, as well as many other aspects of system performance. Brad Chen of Harvard University used modified version of ATOM to instrument the OSF/1 kernel.

11. Acknowledgments

Many people have helped us to bring ATOM to its current form. Jim Keller, Mike Burrows, Roger Cruz, John Edmondson, Mike McCallig, Dirk Meyer, Richard Swan and Mike Uhler were our first internal users, and Dirk Grunwald and Brad Calder provided our first external field test site. Jeremy Dion, Ramsey Haddad, Russell Kao, Greg Lueck, Mike McCallig, and Louis Monier contributed exciting new tools. Many, many others, provided help, support, encouragement, bug reports, flames, and endorsements! Also, Brad Chen, Ted Romer, Ramsey Haddad, Louis Monier, and anonymous USENIX reviewers provided helpful suggestions on the content of this paper. We thank you all.

Appendix: Sample Outputs

The **Read Tool** instruments an application program to determine the total number of bytes read by an application program. A sample output file is shown below.

64916 bytes

In this example, the SPEC92 *nasa*7 benchmark read 64,916 bytes.

The **Profile Tool** instruments an application program to compile a standard instruction profile. A sample output file is shown below.

Procedure	Instructions	Percentage
start	148	0.000
main	107	0.000
compress	58040713	66.487
output	28907385	33.114
rindex	130	0.000
= = •••		
Total	87296502	

For the SPEC92 *compress* benchmark, 28,907,385 instructions were executed inside the output procedure. This represented 33of all the instructions executed in the application program.

The Cache Tool instruments an application program to determine the number of references, misses, and miss rate if the application was run in a 64KB direct mapped, read/write allocate cache with 32 byte lines. A sample output file is shown below.

5387822402 620855884 11.5233

When running the SPEC92 *spice* benchmark, the application would have made 5,387,822,402 references, had 620,855,884 misses, for an overall miss rate of 11.5 percent.

The **Dynamic Memory Tool** instruments an application program to the total amount of memory allocated and freed. A sample output file is shown below.

96107 8208

For the SPEC92 *xlisp* benchmark, 96,107 bytes were allocated and 8,208 bytes were freed.

The Compiler Auditing Tool instruments an application program to find potentially redundant instructions. Here are the results when running the tool on the *ear* benchmark. A sample output file is shown below.

PC	Count	Wasted
0x12000a2d4	188	1
0x12000a4e0	188	2
0x12000a4fc	188	188
0x12000a538	561	188

The instruction at 0x12000a4fc was executed 188 times and never changed the contents of the destination register. This might be worth investigating further.

References

- [1] Anant Agarwal, Richard Sites, and Mark Horwitz. ATUM: A New Technique for Capturing Address Traces Using Microcode. Proceedings of the 13th International Symposium on Computer Architecture, June 1986.
- [2] Robert Bedichek. Some Efficient Architectures Simulation Techniques. Winter 1990 USENIX Conference, January 1990.
- [3] Anita Borg, R.E. Kessler, Georgia Lazana, and David Wall. Long Address Traces from RISC Machines: Generation and Analysis, Proceedings of the 17th Annual Symposium on Computer Architecture, May 1990, also available as WRL Research Report 89/14, Sep 1989.
- [4] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. MIT/LCS/TR-516, MIT, 1991.
- [5] Robert F. Cmelik and David Keppel, Shade: A Fast Instruction-Set Simulator for Execution Profiling. Technical Report UWCSE 93-06-06, University of Washington.
- [6] Susan J. Eggers, David R. Keppel, Eric J. Koldinger, and Henry M. Levy. Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor. SIGMETRICS Conference on Measurement and Modeling of Computer Systems, vol 8, no 1, May 1990.
- [7] Stephen R. Goldschmidt and John L. Hennessy, The Accuracy of Trace-Driven Simulations of Multiprocessors. CSL-TR-92-546, Computer Systems Laboratory, Stanford University, September 1992.
- [8] Reed Hastings and Bob Joyce. Fast Detection of Memory Leaks and Access Errors. Winter 1992 USENIX Conference, January 1992.
- [9] John L. Hennessy and David A. Patterson. Computer Architecture: A Quantitative Approach, pp. 408-425, Morgan Kaufmann, 1990.

- [10] Brian W. Kernighan and Dennis M. Ritchie. The C Programming Language, Prentice-Hall, 1978.
- [11] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. Software, Practice and Experience, vol 24, no. 2, pp 197-218, February 1994.
- [12] James R. Larus and Satish Chandra. Using Tracing and Dynamic Slicing to Tune Compilers. University of Wisconsin Computer Sciences Department Technical Report #1174. August, 1993
- [13] MIPS Computer Systems, Inc. Assembly Language Programmer's Guide, 1986.
- [14] Digital Equipment Corporation. Third Degree Reference Manual, 1993
- [15] Digital Equipment Corporation. ATOM Reference Manual, 1993
- [16] Digital Equipment Corporation. ATOM User Manual, 1993
- [17] Richard L. Sites, ed. Alpha Architecture Reference Manual Digital Press, 1992.
- [18] Amitabh Srivastava and Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation, June, 1994.
- [19] Amitabh Srivastava and David W. Wall. A Practical System for Intermodule Code Optimization at Link-Time. *Journal of Programming Language*, 1(1), pp 1-18, March 1993. Also available as WRL Research Report 92/6, December 1992.
- [20] Amitabh Srivastava and David W. Wall. Link-Time Optimization of Address Calculation on a 64-bit Architecture. Proceedings of the SIG-PLAN'94 Conference on Programming Language Design and Implementation, to appear. Also available as WRL Research Report 94/1, February 1994.

- [21] Amitabh Srivastava. Unreachable procedures in object-oriented programming, *ACM LO-PLAS*, Vol 1, #4, pp 355-364, December 1992. Also available as WRL Research Report 93/4, August 1993.
- [22] David W. Wall. Systems for late code modification. In Robert Giegerich and Susan L. Graham, eds, Code Generation Concepts, Tools, Techniques, pp. 275-293, Springer-Verlag, 1992. Also available as WRL Research Report 92/3, May 1992.

OSF/1 is a registered trademark of the Open Software Foundation, Inc. Alpha, AXP, and DECstation are trademarks of Digital Equipment Corporation.

Alan Eustace received B.S., M.S, and Ph.D. degrees in Computer Science from the University of Central Florida. Since 1986, he has been a researcher at the Digital Equipment Corporation Western Research Laboratory. His interests include VLSI design, computer architecture, and tools for understanding and improving program performance and correctness.

Amitabh Srivastava received a B.Tech. in Electrical Engineering from Indian Institute of Technology, Kanpur, and his M.S. in Computer Science from Pennsylvania State University. Since 1991, he has been a researcher at the Digital Equipment Corporation Western Research Laboratory, working in compilers and linktime code modification. He is the architect of the OM link-time technology which he used to build OM and ATOM systems. Prior to that he was researcher at the Texas Instruments Central Research Labs working on Lisp machines, compilers and object-oriented programming extensions to Scheme. He is the author of the SCOOPS system.

Address for correspondence: Digital Equipment Corporation Western Research Laboratory, 250 University Avenue, Palo Alto, California, 94301, or by electronic mail at eustace@wrl.dec.com or amitabh@wrl.dec.com.

Adaptable Binary Programs

Susan L. Graham

Steven Lucco

Robert Wahbe

Computer Science Division, 571 Evans Hall UC Berkeley, Berkeley CA, 94720

Abstract

To accurately and comprehensively monitor a program's behavior, many performance measurement tools transform the program's executable representation or binary. By instrumenting binary programs to monitor program events, tools can precisely analyze compiler optimization effectiveness, memory system performance, pipeline interlocking, and other dynamic program characteristics that are fully exposed only at this level. Binary transformation has also been used to support software-enforced fault isolation, debugging, machine re-targeting and machine-dependent optimization.

At present, binary transformation applications face a difficult trade-off. Previous approaches to implementing robust transformations result in significant disk space and run-time overhead. To improve efficiency, some current systems sacrifice robustness, relying on heuristic assumptions about the program and recognition of compilerdependent code generation idioms. In this paper we begin by investigating the run-time and disk space overhead of transformation strategies that do not require assumptions about the program's control flow or register usage. We then detail simple information about the binary program that can significantly reduce this overhead. For each type of information, we show how it enables a corresponding type of binary transformation. We call binary programs that contain such enabling information adaptable binaries. Because adaptable binary information is simple, any compiler can generate it. Despite its simplicity, adaptable binary information has the necessary and sufficient expressive

power to support a rich set of binary transformations.

We evaluated a prototype adaptable binary transformation system under the Ultrix 4.2 operating system and the MIPS processor architecture. Using the C spec92 benchmarks, we assessed adaptable binaries in two ways. First, we demonstrated that the information necessary to build adaptable binaries can be compactly recorded, increasing space overhead by only 9% for the spec92 benchmarks. Second, we measured the run-time overhead of previous approaches to implementing robust binary transformations, and showed that adaptable binaries enable a significant reduction of this overhead.

1 Introduction

Program development and monitoring tools are frequently implemented in two parts. The first part instruments a target program, transforming that program to monitor its behavior. The transformed program interacts with the second part of the monitoring tool, a run-time library that records information and takes action in response to events in the transformed program. To accurately and comprehensively monitor a program's behavior, many monitoring tools must instrument

Email: {graham, lucco, rwahbe}@cs.berkeley.edu

This work was supported in part by the National Science Foundation (CDA-8722788), Advanced Research Projects Agency (ARPA) under grant MDA972-92-J-1028 and contracts DABT63-92-C-0026 and N00600-93-C-2481. The content of the paper does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

the program's executable representation or binary. For example, to measure memory system performance, a profiling tool must be aware of register spilling decisions made by the high-level language compiler, and instruction organization decisions made by the linker. At the binary level, all such decisions have been resolved. By transforming binary programs to monitor program events, tools can precisely analyze compiler optimization effectiveness, memory system performance, pipeline interlocking, and other dynamic program characteristics that are fully exposed only at this level.

Instrumentation of programs at the binary level also simplifies the engineering of program monitoring tools. Binary instrumentation can be made compiler-independent, facilitating the analysis of programs written in different high-level languages. Applying transformations to the binary eliminates the complexity and cost of recompilation. Further, by applying transformations to the binary rather than higher-level representations, instrumentation code cannot alter compilation decisions. Finally, binary transformation can be applied to code, such as system libraries, for which source is typically unavailable.

For these reasons, a number of performance analysis tools are implemented as binary transformations [BL92, BKW90, Digc, GH90, Wal91]. For example, pixie calculates instruction frequencies and floating point interlocks based on profile data and the instruction sequence of each basic block [Digc]. QPT uses control flow analysis to support collection and compression of address traces and profiles [BL92]. To aid programmers in identifying memory hierarchy bottlenecks, MTool compares actual and estimated execution times for different regions of a program [GH90]. Borg, Kessler, and Wall use binary transformation to generate and analyze very long multi-program address traces [BKW90].

Binary transformation has also been used to support software-enforced fault isolation, debugging, machine re-targeting and machine-dependent optimization. By inserting code to efficiently monitor indirect control transfers and memory updates, software-enforced fault isolation insures that program errors in one module do not corrupt data in other modules [WLAG93]. Instrumented memory references have been used to implement data breakpoints, detect memory leaks, and trap reads of uninitialized data [HJ92, Cen, WLG93]. Several systems have used binary transformation to re-target a program to a new architecture [HB89, SCK+93, Ech92, BKKM87]. Fi-

nally, optimizations such as code motion, dead-code elimination, register allocation, and instruction scheduling have been applied to binary programs, exploiting the global information available at the binary level [Joh90, SW93, Wal86, Wal92].

At present, binary transformation applications face a difficult trade-off. Robust and compiler-independent techniques for disassembly, insertion of instrumentation code, and obtaining free registers for use by the instrumentation code incur significant disk space and run-time overhead. To improve efficiency, some systems sacrifice robustness, relying on heuristic assumptions about the program's control flow and register usage and recognition of compiler-dependent code generation idioms. This reliance on heuristic information limits the scope and effectiveness of a binary transformation application.

In this paper we begin by investigating the runtime and disk space overhead of transformation strategies that do not require assumptions about the program's control flow or register usage. We then detail simple information about the binary program that can significantly reduce this overhead. We refer to binaries that include this information as adaptable binaries (AB).

Adaptable binaries contain three classes of information. First, control information allows transformation tools to distinguish code from data, identify basic blocks, and identify targets of indirect control transfer instructions. Second, transformation tools can use relocation information to insert, delete, and reorder machine instructions. Finally, register information helps identify free registers that are available for use by instrumentation code.

We have implemented and evaluated adaptable binaries under the Ultrix 4.2 operating system and the MIPS architecture. Using the C spec92 benchmarks, we evaluated adaptable binaries in two ways. First, we measured the additional disk space required by adaptable binaries. We show that the information necessary to support adaptable binaries can be compactly recorded, increasing space overhead by only 9% for the spec92 benchmarks. For comparison, the average space overhead of the standard symbol table included in unstripped executable files is 81%. Debugging information increased executable file size by an average of 123%.

Second, we measured the run-time overhead of previous approaches to implementing robust binary transformations, and showed that adaptable binaries significantly reduce this overhead. These measurements of the basic binary transformation operations show that adaptable binaries can significantly improve the performance of many binary transformation applications.

The rest of the paper is organized as follows. Section 2 defines the fundamental operations required by binary transformation applications and presents the overhead of different robust implementation techniques. Section 3 details the information and program analysis required to build adaptable binaries and discusses our prototype implementation. Section 4 surveys the related work in this area.

2 Implementation Issues

In this section, we present transformation operations required by essentially all binary transformation applications and outline a number of corresponding implementation strategies. We demonstrate that, because existing binaries lack control, relocation and register information, these strategies must rely on conservative assumptions about the program. In the next section we define adaptable binaries, our solution to addressing these issues. Adaptable binaries contain the minimum information required for efficient and robust binary transformation applications.

To quantify the impact of different implementation strategies, we measured the run-time transformation overhead. Transformation overhead is the amount of time spent executing instructions that support the instrumentation code. For example, instructions that save and restore registers in order to obtain them for instrumentation purposes are counted as contributing to the transformation overhead. Neither the time spent in the original program nor the time spent in instrumentation code is part of the transformation overhead. Hence, transformation overhead isolates the runtime cost of supporting binary transformation operations.

Our experiments used the C spec92 benchmarks and were performed on a DecStation 5000/240 with 32 Megabytes of memory. The standard system libraries were not instrumented.

2.1 Disassembly

The fundamental difficulty in disassembling programs is reliably distinguishing code from data. A number of compilation environments place data in the binary's code segment. Under some object formats, since only the code segment is made

read-only and sharable, it is a natural place for read-only data such as program constants. Unfortunately, many systems continue this practice even under object formats that provide appropriate types of data segments. If data is mistaken for code, the transformed program will be incorrect. If code is mistaken for data, parts of the program will not be instrumented. This problem is exacerbated on architectures with variable length instructions; code sequences are not constrained to begin on word boundaries, making disassembly dependent on the assumed starting point of the code.

In the presence of indirect control transfers, there is no robust method for distinguishing data from code. Most control transfer instructions have statically resolvable targets; however, the target of an indirect control transfer instruction is specified via a register and can only be determined at runtime. Common programming language abstractions such as function pointers, case statements, and continuations are typically implemented using indirect control transfer instructions.

On architectures with word-aligned fixed length instructions, a clever and robust solution to the problem of data in the code segment, first employed by pixie, is to duplicate the code segment, and instrument only the duplicated code segment. The duplicated code segment is assumed to contain only code and thus some data may be instrumented. Except for immediates in control transfer instructions, all addresses are left unaltered. Thus, all load and store addresses will reference data in the original code segment. Indirect control transfer instructions, on the other hand, must have their addresses dynamically relocated to the instrumented code segment. Dynamic relocation uses a translation table, built at transformation time, that maps old target addresses to new target addresses. The relocation table is the same size as the original code segment. Why not just leave addresses used to access data in the text segment unaltered and adjust addresses used for control transfers? Because it is not possible, without the use of heuristics, to determine at transformation time whether an address is going to be used as a data address or as a control address. The addition of the duplicated code segment and the dynamic relocation table triples the size of the bi-

Figure 1 gives the code sequence used to perform dynamic relocation. The extra memory access instructions are necessary to obtain a scratch register. We inserted this code sequence before

Benchman	·k	DYN-RELOC	OUT-OF-LINE
052.alvinn	FP	55%	112%
026.compress	INT	18%	104%
056.ear	FP	28%	98%
023.eqntott	INT	20%	86%
008.espresso	INT	10%	102%
001.gcc1.35	INT	58%	154%
022.li	INT	49%	170%
072.sc	INT	34%	71%
Average		34%	112%

Table 1: Transformation overhead, relative to native execution time, for dynamic relocation and out-of-line insertion strategies for inserting instrumentation code before each load and store in the program.

Memory ← temp-reg

Save temporary register temp-reg.

temp-reg ← target-addr - code-start-addr

Calculate offset of current
jump target address.

temp-reg ← temp-reg + table-base-addr

Add jump target address offset to
base of dynamic relocation table.

target-reg ← Memory [temp-reg]

Load new jump target
address from relocation table.

temp-reg ← Memory

Restore temporary register temp-reg.

Figure 1: Assembly pseudo code for dynamic relocation. Without register information to determine which registers are live, it is necessary to save and restore a temporary register. If the code-start-addr and table-base-addr are known at transformation time, one arithmetic instruction could be saved.

every indirect control transfer in the original version of each of the programs listed in Table 1. The DYN-RELOC column of Table 1 shows that the average performance overhead of this approach was 34% on our example programs.

On architectures with variable length instructions, such as the Intel x86, the solution employed by pixie does not work, since correct disassembly depends on the assumed starting point of each code fragment. Robust transformations on architectures with variable length instructions require dynamic disassembly. The transformation system must monitor indirect control transfer instructions, performing dynamic disassembly at previously unseen target addresses. In a study by Cmelik and Keppel, overhead for dynamic disassembly on the SPARC architecture was approximately 600% for a small set of C programs [CK93].

2.2 Edit Operations

Edit operations are used to insert, delete, and reorder machine instructions. These operations are fundamental to all binary transformation applications, since by definition such applications modify the program to alter or monitor its behavior. Edit operations may change the addresses of instructions and data objects, requiring that all references to these objects be updated. Computed addresses and instruction addresses stored in the data segment cannot be reliably identified and updated statically. For example, a common implementation strategy for case statements is to use a jump table stored in the data segment. The jump table stores the code address for each arm of the case statement. There is no reliable way to distinguish a jump table from other kinds of data.

A simple solution to this problem, employed by

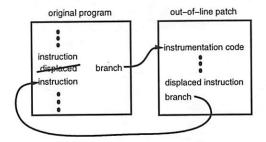


Figure 2: Simple example of out-of-line insertion of instrumentation code.

a number of applications [Digc, LB92, Wal91], is to use the dynamic relocation technique described above. As shown in Table 1, the average run-time overhead of this approach was 34% on our example programs. In addition to the space and run-time overhead of dynamic relocation, system that employ dynamic relocation must deal with indirect function calls from uninstrumented system codes. For example, the signal system call in UNIX takes the address of a signal handler. Since dynamic relocation necessarily leaves addresses unaltered until translation at the indirect control site, the address passed to the signal system call will be the old address of the signal handler. Since the signal code can not be instrumented, no dynamic relocation is performed and the application will have incorrect behavior.

As an alternative to directly inserting instrumentation code, a binary transformation tool can use out-of-line insertion to transfer control to instrumentation code without altering existing instruction addresses [Kes90]. To logically insert instrumentation code before a particular instruction, the instruction is replaced with a control transfer instruction to the instrumentation code. Before returning to the original instruction stream, the displaced instruction is executed. Figure 2 depicts a simple example of out-of-line insertion.

The OUT-OF-LINE column of Table 1 gives the transformation overhead for inserting instrumentation code before every load and store instruction. On average, the OUT-OF-LINE incurs 112% execution time overhead.

2.3 Register Operations

Instructions inserted by a binary transformation tool may need to use machine registers to compute intermediate values or memory addresses. We call such registers temporary registers because they are not live across instructions from the original program. Some applications, such as the memory system simulation program MemSpy, require a large number of temporary registers [MGA92]. In addition, many binary transformation applications can benefit from registers reserved for their exclusive use. For example, pixie uses a reserved register to hold the base of its dynamic relocation table. Wahbe, Lucco, and Graham show that reserving registers for a data breakpoint facility can significantly reduce run-time overhead [WLG93].

Temporary registers are simple to obtain. On entry to the instrumentation code the required registers are saved; these registers are restored upon exit from the instrumentation code.

To reduce the run-time penalty of obtaining temporary registers, a binary transformation tool can use live register information to avoid saving and restoring registers whose values are no longer needed by the original program. However, calculating live register information requires an accurate control flow graph, not possible in the presence of unknown indirect control transfers. Further, without inter-procedural register usage information, transformation applications must assume that all registers not mentioned in the procedure are live, that procedure calls use and define all registers, and that all registers are live on exit from the procedure.

Table 2 shows the cost of obtaining 2, 4, and 8 temporary registers before every load and store instruction in our benchmark programs with and without register usage information. For example, Table 2 shows that the execution time overhead of obtaining 4 registers with accurate register usage information is only 3%, compared to 157% without such information.

Finally, allocating reserved registers also can introduce significant run-time overhead. All uses of the reserved register in the original program must be removed. This requires either obtaining other free registers or mapping the reserved register to a fixed memory location. Because programs can manipulate the stack in unpredictable ways (such as allocating space using the C library call alloca), the stack can not be used to hold register values, further complicating reserved register allocation.

3 Building Adaptable Binaries

In this section, we define the minimum information required for efficient and robust support of

ersprendigen i g		Without Register Information			With Register Information		
Benchmark		Two	Four	Eight	Two	Four	Eight
an musesas		Registers	Registers	Registers	Registers	Registers	Registers
052.alvinn	FP	44%	91%	199%	1%	1%	13%
026.compress	INT	43%	83%	311%	2%	3%	12%
056.ear	FP	63%	133%	289%	0%	0%	4%
023.eqntott	INT	69%	153%	337%	0%	1%	8%
008.espresso	INT	54%	281%	668%	2%	4%	14%
001.gcc1.35	INT	51%	91%	195%	3%	9%	19%
022.li	INT	68%	310%	892%	3%	5%	11%
072.sc	INT	45%	114%	289%	1%	3%	12%
Average		55%	157%	398%	2%	3%	12%

Table 2: Transformation overhead, relative to native execution time, of obtaining the specified number of scratch registers at each load and store instruction.

disassembly, edit, and register operations on binary programs. An adaptable binary is any executable program representation that contains this information. We also outline the implementation of our adaptable binary system, which uses this information to support binary transformation.

3.1 Adaptable Binary Information

The necessary information falls into three categories. Control information provides enough information to construct a control flow graph for each procedure, providing support for distinguishing code from data and identifying basic blocks. Relocation information supports editing operations by allowing references to instruction and data objects to be statically updated. Finally, register usage information, in conjunction with the control information, supports live register analysis, significantly improving the performance of register operations.

This information can not be derived reliably from current binary programs. Fortunately, the necessary information is readily available in most compilers and can be compactly recorded using conventional binary symbol tables. To conserve space, only information that is impossible to derive is stored in the binary; our adaptable binary system reconstructs complete control, relocation, and register usage information at transformation time.

3.2 Control Information

Three components of the control flow graph can not be derived reliably and are maintained in the adaptable binary:

- The beginning and ending address for each procedure.
- Entry addresses for each procedure.
- The possible targets of indirect control transfers.

Indirect control transfer targets are specified using target groups. Target groups are named sets of addresses; an address may belong to any number of target groups, and target groups need not be unique. Target groups are used for two reasons. First, because many indirect control transfers have the same set of possible target addresses (e.g. procedure returns), target groups provide considerable space savings. Second, target groups are used to support separate compilation. The targets for certain classes of indirect control transfers. for example exception handling, might reside in files not yet processed by the compiler. Because new addresses can be added to target groups at any time, they provide a convenient level of indirection in naming target addresses.

Given the above control information, our adaptable binary system locates basic blocks using the following algorithm:

1. Initialize work list to entry address of the program¹. Repeat steps 2 and 3 until the work list is empty.

¹Since the operating system requires the entry address to begin program execution, we can safely assume that it is specified in all binaries.

- Remove an address address from the work list and create a corresponding basic block or split an existing block. Add instructions to the current basic block, beginning at address, until either a control flow instruction is encountered or there are no more instructions.
- 3. Given a control flow instruction, if any of the target addresses have not been processed as in Step 2, add the addresses to the work list.

Unlike conventional algorithms that linearly process the program to discover basic blocks [ASU86], the above algorithm guarantees that no data is inadvertently processed as code. Unreachable areas in the code segment are simply treated as data. Given the basic blocks, the next step is to build the control flow graph.

On architectures without delayed control transfers, building the control flow graph is a simple task [ASU86]. In the presence of branch delay slots, especially annulled delay slots, building the control flow graph is more difficult than for higher-level language programs, but still straightforward given the adaptable binary information [LB92].

3.3 Relocation Information

Edit operations, such as inserting instrumentation code, can change the address of instructions and data objects. Adaptable binaries must provide information to update these references at transformation time, eliminating the need for techniques that incur run-time overhead, such as dynamic relocation or out-of-line code insertion.

Traditional linkers combine one or more object files into a single executable, relying on relocation information to locate and update all program references. Each relocation entry consists of a pointer to the affected reference and the operations required to update it. Traditionally, the linker discards the relocation information following creation of the binary. Adaptable binaries retain this standard *inter-file* relocation information.

In theory, inter-file relocation is not sufficient to support transformations that change the relative order of instructions within a file. For example, consider the indirect control transfer depicted in Figure 3. Because conventional interfile relocation information assumes that the relative placement of instructions within a file remains unchanged, support for updating the reference CodeLabel is provided, but not, typically, the constant offset -16. If instrumentation code is

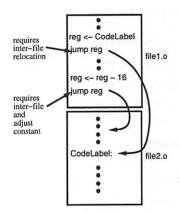


Figure 3: Two indirect control transfer instructions. The first jump requires only inter-file relocation information. Because the second jump's target address is computed using an offset (-16) relative to a relocated label (CodeLabel), inter-file relocation is not sufficient.

inserted between CodeLabel and CodeLabel-16, the constant offset must be adjusted.

As another example where inter-file relocation is not sufficient, consider Figure 4. If the size of the instrumentation code placed before CodeLabel1 is different than the instrumentation code placed before CodeLabel2, there is no easy way to adjust the offset -16.

As a rule, adaptable binaries must contain sufficient relocation information necessary to update addresses if the relative position of instructions is changed. To date, we are not aware of any binaries that make use of computed target addresses as described above. Hence, rather than complicate the relocation information, we simply prohibit adaptable binaries from using these types of computed addresses.

3.4 Register Usage Information

As outlined in Section 2, register operations that allocate temporary and reserved registers can benefit from live register information. For each procedure, adaptable binaries store the following information:

- livereg-gen Registers whose value is used in the the procedure.
- livereg-kill Registers whose value is defined in the procedure.
- livereg-live-on-exit Registers whose value is live on exit to the procedure.

if ()
reg = CodeLabel1
else
reg = CodeLabel2
reg <- reg - 16
jump reg

CodeLabel1:

CodeLabel2:

Figure 4: Example instruction sequence which complicates binary transformation applications. If the size of the instrumentation code inserted before CodeLabel1 and CodeLabel2 is different, there is no easy way to adjust the computed address reg - 16.

In the presence of callee-saved registers, livereggen and livereg-kill can not be precisely derived. Consider the following function:

```
function f()

/* Save callee-saved register before using it. */

Memory ← reg

... function body ...

/* Restore callee-saved register. */

reg ← Memory
```

If there are assignments to memory in f()'s body, conventional program analysis would conclude that f() both uses and defines the value in reg. With respect to the call site, however, the value of reg is preserved across calls to f(), and is thus semantically neither used nor defined. When the compiler emits this register spill code, it can easily and precisely construct livereg-gen and livereg-kill to reflect this state. Because highly optimized programs can violate standard calling conventions, without precise livereg-gen and livereg-kill information, a transformation application using reg would be forced to unnecessarily spill the register before calling f().

3.5 Our Adaptable Binary System

We have built a prototype adaptable binary system. The system supports both ecoff and a out binary formats and can read OSF/1, SunOS, and Ultrix binaries. At present, it only includes disassembly modules for the MIPS and Sparc architectures.

We modified gcc to output adaptable binary (AB) information. Because the AB information was readily available in gcc data structures, adding this functionality to gcc required less than 800 lines of C code and less than a week's work.

3.6 Disk Space

Table 3 presents the disk space overhead of the control, relocation, and register usage information required by adaptable binaries. Our prototype stores adaptable binary information using the conventional binary symbol table and compresses this data using a variation of Liv-Zempel compression [LZ77]. We found that using a standard compression algorithm rather than semantic compression allowed simplified layouts of the adaptable binary information and yielded similar disk space savings. For comparison, we also measured the space overhead of the standard symbol table included in all unstripped executables and the symbol information required during debugging. As can be seen from Table 3, compressed adaptable binary information requires significantly less space than standard symbol table information.

4 Related Work

A number of systems have added information to binaries to facilitate transformation systems. Johnson modified the Stardent linker 1d to retain inter-file relocation information [Joh90]. In addition, Stardent compilers were restricted from performing certain machine-dependent optimizations and placing data in the code segment. The Stardent system did not consider control or register information.

Like Johnson's system, epoxie retains inter-file relocation information [Wal91]. Rather than modify the linker, epoxie can use any linker that supports incremental linking. Incremental linking, (e.g. ld -r), retains relocation information, allowing already combined object files to be repeatedly linked.

The Mahler system performs transformations in the linker, providing transformation applications with inter-file relocation information [Wal92]. Like Stardent's 1d and epoxie, no control information is included in the binary. Mahler performs several optimizations, including global register allocation. To accomplish this, register actions are included which tell the linker how to modify the binary if it decides to promote a variable from memory to a register. Register actions provide, however, only

Benchmark		Adaptable Binary Information	Standard Header	Standard Header + Debug Symbols
052.alvinn	FP	5%	95%	100%
026.compress	INT	8%	94%	107%
056.ear	FP	9%	82%	107%
023.eqntott	INT	9%	91%	131%
008.espresso	INT	9%	47%	151%
001.gcc1.35	INT	9%	92%	124%
022.li	INT	9%	93%	181%
072.sc	INT	11%	50%	81%
Average		9%	81%	123%

Table 3: Disk space overhead for adaptable binary information. For context, we also present the disk space overhead for the standard header included in all unstripped programs, as well as the symbol overhead when program's are compiled for debugging.

limited support for determining live register information.

In addition, systems have used compiler-dependent heuristics to approximate the control, relocation and register information present in adaptable binaries [Wal91, GH90, LB92, SW93]. Control and relocation information is synthesized by pattern matching for compiler-dependent instruction idioms. For example, to find jump tables, MTool searches for the instruction sequence it assumes will implement case statements [GH90]. Larus and Ball exploit register usage conventions when allocating registers, relying on programmer input to discover cases in which these assumptions are invalid [LB92].

Relying on compiler-dependent heuristics reduces the scope and effectiveness of a binary transformation application. For highly optimized code, finding a reliable heuristic might be difficult or impossible. Programs frequently use libraries written in different high-level languages or generated by different compilers. In these cases, different and potentially incompatible heuristics might be necessary. Because there are no constraints on the compiler, it is impossible to insure that the heuristics cover all possible sequences of generated code. For example, the Ultrix C compiler will violate normal MIPS register usage conventions when asked to do global register allocation[Digb]. Finally, a binary transformation tool that relies on compiler-dependent heuristics must be updated and tested with each new release of the compiler.

Adaptable binaries avoid these difficulties by including extra information in the executable representation of a program. Once annotated with this information, a binary program or library can sup-

port a large range of binary transformation applications, regardless of its origin.

On the other hand, support for adaptable binaries requires modifications to existing compilers. Even small changes to compilers, such as the ones described in this paper, may be politically difficult to get adopted. Until compilers export adaptable binary information, binary transformation systems must choose between added efficiency at the cost of compiler-dependent heuristics.

Finally, researchers have performed transformations on higher level program representations. For example, modifying assembly or object files addresses the problem of deriving relocation information. It does not address the need for control and register usage information.

Further, instrumenting a program at higher levels of abstraction poses the problem that many classes of program events are influenced by compilation decisions not fully resolved until the binary is created. For example, instrumenting a compiler's intermediate form can significantly affect the code generated. On many systems, the assembly representation contains high-level pseudo instructions, hiding machine instruction selection. On MIPS systems, the assembler performs machine instruction selection, delay slot filling, and instruction scheduling [Diga]. The relative order of procedures as well as final addresses are not established until the object files are processed by the linker.

5 Conclusion

We have described adaptable binaries, a technique for supporting robust and efficient binary transformation. First, we identified a set of operations that are fundamental to binary transformation. At least one of these operations is necessary for every binary transformation application that we surveyed. Second, we defined the minimum information required for efficient and robust support of these operations. We detailed the subset of this information that can not be derived from current binary programs and demonstrated that this subset of information can be added to executable files with negligible space overhead. Finally, we demonstrated quantitatively that augmenting binary programs with this information can dramatically reduce the run-time overhead of instrumented programs.

Adaptable binaries establish the necessary and sufficient information that any compiler must provide to support the basic binary transformation operations. Once annotated with this information, a binary program or library can support a large range of binary transformation applications, regardless of its origin. We hope that this work will influence compiler-writers and maintainers to make the output of adaptable binary information a standard compiler option.

Authors

Susan L. Graham is a professor of Computer Science at the University of California, Berkeley. Her research interests include programming language implementation, software development environments, and high-performance computing.

Steve Lucco is an assistant professor of Computer Science at Carnegie Mellon University. His research interests include software-enforced fault isolation, new organizations for operating systems, and high-performance computing.

Robert Wahbe is a Ph.D. candidate in Computer Science at the University of California, Berkeley. His current research focuses on mechanisms to allow databases and operating systems to safely incorporate third-party extensions written in traditional programming languages such as C and Fortran.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Compilers, Principles, Techniques, and Tools. Addison-Wesley Publishing Company, 1986.

- [BKKM87] A. Bergh, D. Margenheimer K. Keilman, and J. Miller. HP 3000 emulation on HP precision architecture computers. Hewlett-Packard Journal, December 1987.
- [BKW90] Anita Borg, R.E. Kessler, and David W. Wall. Generation and analysis of very long address traces. In *International Symposium on Computer Architecture*, pages 270-279, May 1990.
- [BL92] Thomas Ball and James R. Larus. Optimally profiling and tracing. In *Proceedings* of the Conference on Principles of Programming Languages, pages 59-70, 1992.
- [Cen] CenterLine Incroporated. TestCenter Manual.
- [CK93] Robert F. Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report UWCSE 93-06-06, University of Washington, 1993.
- [Diga] Digital Equipment Corporation. Ultrix v4.2 as Manual Page.
- [Digb] Digital Equipment Corporation. Ultrix v4.2 cc Manual Page.
- [Digc] Digital Equipment Corporation. *Ultrix* v4.2 pixie Manual Page.
- [Ech92] Echo Logic, Inc. News Release, May 1992.
- [GH90] Aaron Goldberg and John Hennessy.
 MTOOL: A method for detecting memory bottlenecks. Technical Report TN-17,
 Digital System Research Center, December 1990.
- [HB89] C. Hunter and J. Banning. DOS at RISC.

 Byte Magazine, pages 361-368, November 1989.
- [HJ92] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the 1992 Usenix* Winter Conference, pages 125-136, 1992.
- [Joh90] S. C. Johnson. Postloading for fun and profit. In Proceedings of the Winter USENIX Conference, pages 325-330, 1990.
- [Kes90] Peter B. Kessler. Fast breakpoints: Design and implementation. In Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, pages 78-84, White Plains, New York, June 1990. Appeared as SIGPLAN Notices 25(6).
- [LB92] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. Technical Report 1083, University of Wisconsin-Madison, March 1992.

- [LZ77] J. Liv and A. Zempel. A universal algorithm for sequential data compression. IEEE Transactions on Information Theory, 23(3):337-343, 1977.
- [MGA92] Margaret Martoonosi, Anoop Gupta, and Thomas Anderson. MemSpy: Analyzing memory system bottlenecks in programs. In Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 1-12, 1992.
- [SCK+93] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. Communications of the ACM, 36(2):69-81, February 1993.
- [SW93] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, March 1993.
- [Wal86] David W. Wall. Global register allocation at link time. In Proceedings of the Conference on Programming Language Design and Implementation, pages 264-275, July 1986. Appeared as SIGPLAN NOTICES 21(7).
- [Wal91] David W. Wall. Systems for late code modification. Technical Report TN-19, Digital Western Research Laboratory, June 1991.
- [Wal92] David W. Wall. Experience with a software-defined machine architecture. ACM Transactions on Programming Languages and Systems, 14(3), July 1992.
- [WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In Proceedings of the Symposium on Operating System Principles, pages 203-216, December 1993.
- [WLG93] Robert Wahbe, Steven Lucco, and Susan L. Graham. Practical data breakpoints: Design and implementation. In Proceedings of the Conference on Programming Language Design and Implementation, pages 1-12, 1993. Appeared as ACM Sigplan Notices 28 (6).

NOTES

The USENIX Association

Founded in 1975, the USENIX Association is the original not-for-profit professional and technical organization for individuals and institutions with an interest in UNIX and advanced computing systems. The USENIX Association and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and rapid communication of results
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

USENIX holds an annual technical conference, an annual system administration conference co-sponsored with SAGE, and single topic symposia throughout the year. It publishes ;login:, a bi-monthly newsletter; Computing Systems, a quarterly technical journal published in association with The MIT Press; and Conference Proceedings for each of its conferences and symposia. It also sponsors local and special technical groups relevant to the UNIX environment as well as participating in various standards efforts.

SAGE

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX. SAGE activities include the publishing of *Job Descriptions for System Administrators*, edited by Tina Darmohray; "SAGE News", a regular feature in ;login:, The System Administrator Profile, an annual survey of system administrator salaries and responsibilities; support of working groups; and an archive site for papers from the System Administration Conferences.

Member Benefits:

- Free subscription to ; login:, the Association's bi-monthly newsletter featuring technical articles, a worldwide calendar of events, SAGE News, media reviews, summaries of conferences, Snitch Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts, and much more.
- Free subscription to Computing Systems, a refereed technical quarterly published with The MIT Press.
- Discounts on registration for technical sessions at all USENIX conferences and symposia.
- Discounts on proceedings from USENIX conferences and symposia.
- Discount on the new 4.4BSD Manuals and CD-ROM, the definitive release of the Berkeley version of UNIX published by the USENIX Association and O'Reilly & Associates, Inc.
- Savings on *The Evolution of C++: Language Design in the Marketplace of Ideas*, edited by Jim Waldo of Sun Microsystems Laboratories, the USENIX Association book published by The MIT Press
- Special subscription rates to *UniForum Monthly*, *UniNews*, the annual UniForum Open Systems Products Directory, and UNIX World's *Open Computing*.
- Savings on selected titles from McGraw-Hill, The MIT Press, Prentice Hall, John Wiley & Sons, O'Reilly and Associates, and UniForum.

Supporting Members of the USENIX Association:

ANDATACO
Frame Technology Corporation
Graphon Corporation
Matsushita Electrical Industrial Co., Ltd.

Quality Micro Systems, Inc. Sun Microsystems, Inc., Sunsoft Network Products Tandem Computers, Inc. UUNET Technologies, Inc.

SAGE Supporting Member:

Enterprise System Management Corporation

For further information about USENIX and SAGE, please contact:

The USENIX Association 2560 Ninth Street, Suite 215 Berkeley, CA 94710 USA Telephone: +1 510 528 8649 Email: office@usenix.org

FAX: +1 510 548 5738

WWW URL: http://www.usenix.org